INSTITUT DU DÉVELOPPEMENT ET DES RESSOURCES EN INFORMATIQUE SCIENTIFIQUE

Langage Fortran (Avancé)

Patrick Corde
Hervé Delouis

Patrick.Corde@idris.fr

26 mars 2021

Table des matières I

Table des matières II

Table des matières III

Table des matières IV

Table des matières V

Table des matières VI

- `Code machine` (notation numérique en octal) ;
- `Assembleurs` de codes mnémoniques ;
- 1954 : projet création du premier `langage symbolique` FORTRAN par John Backus d'`IBM` (*Mathematical **FOR**mula **TRAN**slating System*) :
  - Efficacité du code généré (performance) ;
  - Langage quasi naturel pour scientifiques (productivité, maintenance, lisibilité).
- 1957 : Livraison des premiers compilateurs ;
- 1958 : **Fortran II** `(IBM)` ⇒sous-programmes compilables de façon indépendante.
- Généralisation aux autres constructeurs mais :
  - divergences des extensions ⇒nécessité de **normalisation** ;
  - `ASA` *American Standards Association* (`ANSI` *American Nat. Standards Institute*). Comité chargé du développement d'une norme Fortran.
- 1966 : **Fortran IV** (Fortran 66) ;
- Évolution par extensions divergentes. . .
- 1977 : **Fortran V** (Fortran 77).
  quasi compatible :
  aucune itération des boucles *nulles* (`DO I=1,0`)
  - **Nouveautés principales** :
    - type caractère ;
    - `IF-THEN-ELSE` ;
    - `E/S` accès direct et `OPEN`.

- Travail des comités `X3J3/ANSI` et `WG5/ISO` pour moderniser Fortran 77 :
  - Standardisation : inclusion d'extensions ;
  - Développement : nouveaux concepts déjà exploités par langages plus récents `APL`, `Algol`, `PASCAL`, `Ada` ; ...
  - Performances en calcul scientifique ;
  - Totalement compatible avec Fortran 77.
- 1991/1992 : Norme **Fortran 90** (`ISO` et `ANSI`) ;
- 1994 : Premiers compilateurs Fortran 90 `Cray` et `IBM` ;
- 1997 : Norme **Fortran 95** (`ISO` et `ANSI`) ;
- 1999 : Premiers compilateurs Fortran 95 sur `Cray T3E` puis `IBM RS/6000` ;
- septembre 2004 : Norme **Fortran 2003** (`ISO` et `ANSI`) ;
- octobre 2010 : Norme **Fortran 2008** (`ISO` et `ANSI`).
- novembre 2018 : norme **Fortran 2018** (`ISO` et `ANSI`).

Compatibilité norme 77/90

- La norme 77 est totalement incluse dans la norme 90.
- Quelques comportements différents :
  - beaucoup plus de fonctions/sous-progr. intrinsèques ⇒ risque d'homonymie avec procédures externes Fortran 77 et donc de résultats différents ! `EXTERNAL` recommandé pour les procédures externes non intrinsèques,
  - attribut `SAVE` automatiquement donné aux variables initialisées par l'instruction `DATA` (en Fortran 77 c'était « constructeur dépendant ») ;
  - E/S - En lecture avec format, si *Input list* > *Record length* (ou plus exactement si une fin d'enregistrement est atteinte avant la fin de l'exploration du format associé à la valorisation de l'*input list*) :
    - OK en Fortran 90 car au niveau de l'`OPEN`, `PAD="YES"` pris par défaut.
    - Erreur en Fortran 77 !

Apports de Fortran 90

- Procédures internes (`CONTAINS`), **modules** (`USE`) ;
- « Format libre », identificateurs, déclarations ;
- Précision des nombres : `KIND` ⇒ portabilité ;
- *Objets* de types dérivés ;
- `DO-END DO, SELECT CASE, WHERE` ;
- Extensions tableaux : profil, conformance, manipulation, fonctions ;
- Allocation dynamique de mémoire (`ALLOCATE`) ;
- Pointeurs ;
- Arguments : `OPTIONAL, INTENT, PRESENT`. Passage par mot-clé ;
- Bloc interface, interface générique, surcharge d'opérateurs ;
- Procédures récursives ;
- Nouvelles fonctions intrinsèques ;
- Normalisation directive `INCLUDE`.

Apports de Fortran 95

- `FORALL(i=1:n,j=1:m,y(i,j)/=0.) x(i,j)=1./y(i,j)`
  (cf. Annexe C page 321).
- Les attributs `PURE` et `ELEMENTAL` pour des procédures sans effet de bord et pour le second des arguments muets élémentaires mais appel possible avec arguments de type tableaux
  (cf. Annexe C pages 317, 319).
- Fonction intrinsèque `NULL()` pour forcer un pointeur à l'état nul y compris lors de sa déclaration (cf. chap. 7 page 153).
- Libération automatique des tableaux dynamiques locaux n'ayant pas l'attribut `SAVE` (cf. chap. 6 page 133).
- Valeur initiale par défaut pour les composantes d'un type dérivé
  (cf. chap. 3 page 59).
- Fonction intrinsèque `CPU_TIME` (cf. chap. 14 page 249).
- Bloc `WHERE` : imbrication possible (cf. chap. 5 page 123).
- Expressions d'initialisation étendues (cf. chap. 5 page 123).
- `MAXLOC`/`MINLOC` : ajout argument `dim` (cf. Chap. 5 page 93).
- Ajout `generic_spec` (cf. Chap. 9 page 197) au niveau de l'instruction `END INTERFACE [generic_spec]`

bibliographie

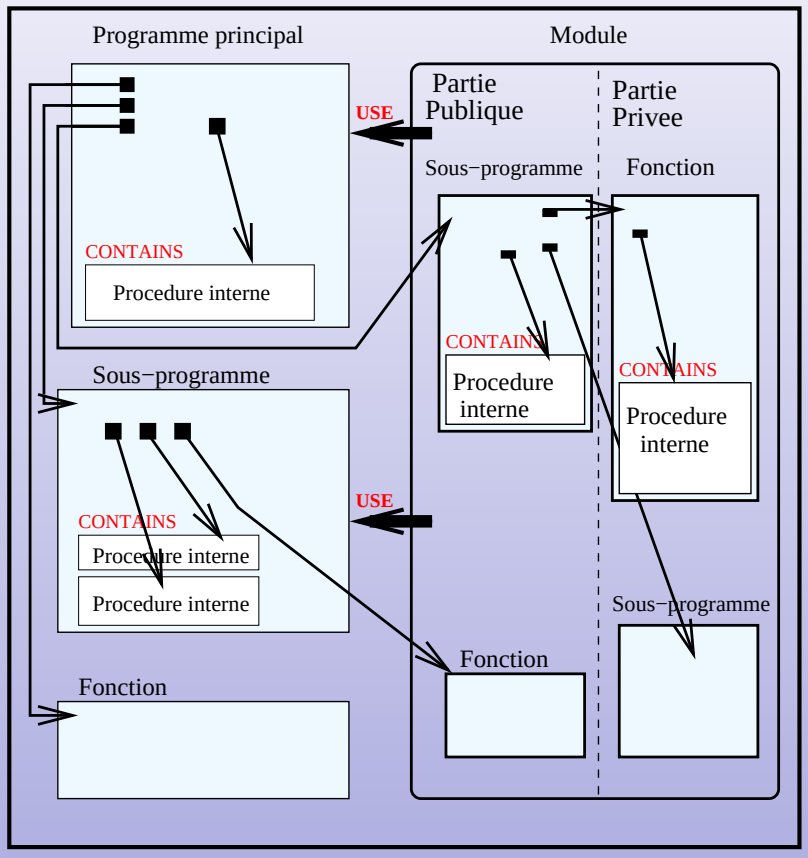
- Adams, Brainerd, Hendrickson, Maine, Martin, Smith, *The Fortran 2003 Handbook*, Springer, 2009, (712 pages), ISBN 978-1-84628-378-9 ;
- Walters S. Brainerd, *Guide to Fortran 2008 Programming*, Springer, 2015, ISBN 978-1-4471-6758-7 ;
- Chivers Ian, Sleightholme Jane, *Introduction to Programming with Fortran*, 2015, ISBN 978-3-319-17701-4 ;
- Michael Metcalf, John Reid, Malcom Cohen, *Modern Fortran Explained*, 2018, ISBN 978-0-19-881189-3 ;
- Norman S. Clerman, Walter Spector, *Modern Fortran : Style and Usage*, Cambridge University Press, 2012, ISBN 978-0521730525 ;
- Arjen Markus, *Modern Fortran in Practice*, Cambridge University Press, juin 2012, (272 pages), ISBN 978-1-10760-347-9 ;
- Chamberland Luc, *Fortran 90 : A Reference Guide*, Prentice Hall, ISBN 0-13-397332-8 ;
- Delannoy Claude, *Programmer en Fortran 90 – Guide complet*, Eyrolles, 1997, (413 pages), ISBN 2-212-08982-1 ;
- Dubesset M., Vignes J., *Les spécificités du Fortran 90*, Éditions Technip, 1993, (400 pages), ISBN 2-7108-0652-5 ;
- Ellis, Phillips, Lahey, *Fortran 90 Programming*, Addisson-Wesley, 1994, (825 pages), ISBN 0-201-54446-6 ;

- Kerrigan James F., *Migrating to Fortran 90*, O'Reilly & Associates Inc., 1994, (389 pages), ISBN 1-56592-049-X ;
- Lignelet P., *Fortran 90 : approche par la pratique*, Éditions Studio Image (série informatique), 1993, ISBN 2-909615-01-4 ;
- Lignelet P., *Manuel complet du langage Fortran 90 et Fortran 95*, calcul intensif et génie logiciel, Col. Mesures physiques,Masson, 1996, (320 pages), ISBN 2-225-85229-4 ;
- Lignelet P., *Structures de données et leurs algorithmes avec Fortran 90 et Fortran 95*, Masson, 1996, (360 pages), ISBN 2-225-85373-8 ;
- Morgan and Schoenfelder, *Programming in Fortran 90*, Alfred Waller Ltd., 1993, ISBN 1-872474-06-3 ;
- Michael Metcalf, John Reid, Malcom Cohen,
  - Fortran 90 explained, Science Publications, Oxford, 1994, (294 pages), ISBN 0-19-853772-7, Traduction française par Pichon B. et Caillat M., Fortran 90 : les concepts fondamentaux, Éditions AFNOR, 1993, ISBN 2-12-486513-7 ;
  - Fortran 95/2003 explained, Oxford University Press, 2004, (416 pages), ISBN 0-19-852693-8 ;
  - *Modern Fortran Explained*, 2011, ISBN 978-0-19-960142-4.

- Olagnon Michel, Traitement de données numériques avec Fortran 90, Masson, 1996, (364 pages), ISBN 2-225-85259-6
- Redwine Cooper, *Upgrading to Fortran 90*, Springer, 1995, ISBN 0-387-97995-6 ;
- *International Standard* ISO/IEC 1539-1:1997(E) *Information technology - Progr. languages - Fortran - Part1 : Base language*. Disponible auprès de l'AFNOR.

- Documentation générale
  - Supports de cours Fortran IDRIS : http://www.idris.fr, choix « Supports de cours » puis « Fortran »
  - *The Fortran Company* : http://www.fortran.com
  - État d'avancement de l'intégration de la norme Fortran 2003 : http://fortranwiki.org/fortran/show/Fortran+2003+status
  - État d'avancement de l'intégration de la norme Fortran 2008 : http://fortranwiki.org/fortran/show/Fortran+2008+status

Structure d'un programme



Compilation, édition des liens, exécution

- Le compilateur crée pour chaque fichier source :
  - un fichier objet de même nom suffixé par `.o`,
  - autant de fichiers *nom_module*`.mod` qu'il y a de modules (sur `IBM RS/SP6`, la commande `what` permet de savoir, entre autres, de quel fichier source ils sont issus).
- Si un module fait appel (`USE`) à d'autres modules, ces derniers doivent avoir été précédemment compilés.

---

1. Compilation **préalable** des sources contenant les modules :
   ```
   f90 -c mod1.f90 mod2.f90
   ```
2. Compil./*link* de `prog.f90` utilisant ces modules :
   ```
   f90 prog.f90 mod1.o mod2.o
   ```
   les fichiers `.mod` (contenant la partie *descripteur*) sont automatiquement trouvés s'ils se trouvent dans le répertoire courant ou dans celui du source. L'option `-I` permet de spécifier d'autres répertoires de recherche prioritaires.
3. Exécution : `a.out`

Les identificateurs

Un identificateur est formé d'une suite de caractères choisis parmi les **lettres** (non accentuées), les **chiffres** et le **blanc souligné**. Le premier d'entre eux doit être obligatoirement une lettre.
La longueur d'un identificateur est limitée à 63 caractères.
On ne distingue pas les **majuscules** des **minuscules**.

Attention : en « format libre », les blancs sont significatifs.

Exemple d'identificateurs

- `compteur`
- `Compteur`
- `fin_de_fichier`
- `montant_annee_1993`

En **Fortran** il existe un certain nombre de **mots-clés** (**real**, **integer**, **logical**, ...), mais qui ne sont pas réservés comme dans la plupart des autres langages. On peut donc, dans l'absolu, les utiliser comme identificateurs personnels. Cependant, pour permettre une bonne lisibilité du programme on évitera de le faire.

Le « format libre »

1. Dans le mode « **format libre** » les lignes peuvent être de longueur quelconque à concurrence de 132 caractères ;
2. Il est également possible de coder plusieurs instructions sur une même ligne en les séparant avec le caractère « **;** » ;
3. Une instruction peut être codée sur plusieurs lignes : on utilisera alors le caractère « **&** » ;
4. Lors de la coupure d'une constante chaîne de caractères la suite de la chaîne doit obligatoirement être précédée du caractère « **&** ».

Exemples

```
print *, " Entrez une valeur :"; read *,n
print *, "Montant  HT :", montant_ht,  &
         "       TVA :", tva       ,  &
         "Montant TTC :", montant_ttc
print *, "Entrez un nombre entier &
        &compris entre 100 & 199"
```

Les commentaires

Le caractère « **!** » rencontré sur une ligne indique que ce qui suit est un commentaire. On peut évidemment écrire une ligne complète de commentaires : il suffit pour cela que le 1er caractère non blanc soit le caractère « **!** ».

Exemple

```
if (n < 100 .or. n > 199) then ! Test cas d'erreur
  . . . .
! On lit l'exposant
read *,x
! On lit la base
read *,y
if (y <= 0) then    ! Test cas d'erreur
    print *," La base doit être un nombre >0"
else
    z = y**x        ! On calcule la puissance
end if
```

Notez la nouvelle syntaxe possible des opérateurs logiques :
.LE. ⇒ <= .LT. ⇒ < .EQ.⇒ ==
.GE. ⇒ >= .GT. ⇒ > .NE.⇒ /=
Les opérateurs .AND., .OR., .NOT. ainsi que .EQV. et .NEQV. n'ont pas d'équivalents nouveaux.

Par contre, il n'est pas possible d'insérer un commentaire entre deux instructions situées sur une même ligne. Dans ce cas la 2e instruction ferait partie du commentaire.

Exemple

```
      i=0 ! initialisation ; j = i + 1
```

Attention :

```
     C----- Commentaire Fortran 77
     c----- Commentaire Fortran 77
     *----- Commentaire Fortran 77
```

ne sont pas des commentaires Fortran en « **format libre** » et génèrent des erreurs de compilation.

Le « format fixe »

Le « format fixe » de **Fortran** correspond à l'ancien format du **Fortran 77** avec deux extensions :

- plusieurs instructions possibles sur une même ligne ;
- nouvelle forme de commentaire introduite par le caractère « ! ».

Son principal intérêt est d'assurer la compatibilité avec **Fortran 77**. C'est un aspect **obsolète** du langage !

Structure d'une ligne en « format fixe » :

- zone étiquette (colonnes 1 à 5)
- zone instruction (colonnes 7 à 72)
- colonne suite (colonne 6)

Les lignes qui commencent par **C, c, *** ou **!** en colonne 1 sont des commentaires.

Les déclarations

Déclarer une variable permet notamment de lui attribuer un type. De même que l'ancien **Fortran** un type par défaut est appliqué aux variables non déclarées. Il est vivement recommandé de spécifier l'instruction `IMPLICIT NONE` en tête de toute unité de programme pour être dans l'obligation de les déclarer. La syntaxe d'une déclaration est la suivante :

```
type[, liste_attributs :: ] liste_identificateurs
```

Liste des différents types :

- **real**
- **integer**
- **double precision**
- **complex**
- **character**
- **logical**
- **type**

Différents attributs :

`parameter`	constante symbolique
`dimension`	taille d'un tableau
`allocatable`	objet dynamique
`pointer`	objet défini comme pointeur
`target`	objet accessible par pointeur (cible)
`save`	objet statique
`intent`	vocation d'un argument muet
`optional`	argument muet facultatif
`public` ou `private`	visibilité d'un objet défini dans un module
`external` ou `intrinsic`	nature d'une procédure

Exemple de déclarations

```
integer nbre, cumul
real x, y, z
integer, save :: compteur
integer, parameter :: n = 5
double precision a(100)
double precision, dimension(100) :: a
complex, dimension(-2:4, 0:5) :: c
real, pointer, dimension(:) :: ptr
real, pointer :: ptr(:) ! non recommandé : pourrait être interprété
                        ! à tort comme un tableau de pointeurs.
```

Note : il est toujours possible de donner le type et les différents attributs d'un objet à l'aide de plusieurs instructions, mais on préfèrera la dernière syntaxe :

Exemple

```
integer tab
dimension tab(10)
save tab

integer, dimension(10), save :: tab
```

Il est possible d'initialiser un objet au moment de sa déclaration. C'est d'ailleurs obligatoire si l'attribut `parameter` est spécifié.

Exemple

```
integer, parameter :: n = 4
character(len=n), dimension(5) :: notes = &
     (/ "do# ", "re  ", "mi  ", "fa# ", "sol#" /)
character(len=n), dimension(5) :: notes = &
     (/ character(len=n) :: "do#", "re", "mi", "fa#", "sol#" /)
integer,dimension(3) :: t_entiers=(/ 1, 5, 9 /)
```

Remarques

- en **Fortran 77** toute variable initialisée (via l'instruction `DATA`) n'est permanente que si l'attribut `save` a été spécifié pour cette variable, ou bien si la compilation a été faite en mode « **static** » ;
- depuis la norme **Fortran 90** toute variable initialisée est permanente : elle reçoit l'attribut `save` implicitement ;
- la norme **Fortran 2008** précise que toute variable déclarée (avec ou sans initialisation) au sein de l'unité de programme principale ou d'un module reçoit l'attribut `save` implicitement.

Typage des données : paramètre KIND

Les types prédéfinis en **Fortran** (depuis la norme 90) sont des noms génériques renfermant chacun un certain nombre de **variantes** ou **sous-types** que l'on peut sélectionner à l'aide du paramètre `KIND` lors de la déclaration des objets. Ce paramètre est un **mot-clé** à valeur entière qui désigne la **variante** souhaitée pour un **type** donné.

Remarques

- les différentes valeurs du paramètre `KIND` sont dépendantes du système utilisé. Elles correspondent, en général, au nombre d'octets désirés pour coder l'objet déclaré ;
- à chaque type correspond une **variante** par défaut, sélectionnée en l'absence du paramètre `KIND` : c'est la simple précision pour les réels.
  (Voir tableau en annexe **A** page 263) ;
- concernant les chaînes de caractères, cette valeur peut indiquer le nombre d'octets utilisés pour coder chaque caractère :
  - 2 octets seront nécessaires pour coder les idéogrammes des alphabets chinois ou japonais ;
  - 1 seul octet suffit pour coder les caractères de notre alphabet.

Exemple

```
real(kind=8) [1] x    ! généralement la double précision.
integer(kind=2) [1], target, save :: i
```

1. cette syntaxe remplace celle utilisée en **Fortran 77** : real∗8, integer∗2

Typage des données : paramètre KIND (nouveautés norme 2008)

La norme 2008 a défini un module (ISO_FORTRAN_ENV) renfermant diverses constantes symboliques facilitant la déclaration des données de type intrinsèque (précision du gabarit mémoire, variantes disponibles, ...) :

1. INTEGER_KINDS, REAL_KINDS, LOGICAL_KINDS sont des tableaux contenant les valeurs des différentes variantes disponibles pour les types INTEGER, REAL et LOGICAL respectivement ;
2. INT8, INT16, INT32 et INT64 permettent de préciser le gabarit (8, 16, 32, 64 bits) d'un entier ;
3. REAL32, REAL64 et REAL128 permettent de préciser le gabarit (32, 64, 128 bits) d'un réel ;
4. NUMERIC_STORAGE_SIZE est la valeur en bits d'une donnée déclarée de type INTEGER, REAL ou LOGICAL par défaut, c-à-d sans spécification du paramètre KIND (32 généralement) ;
5. CHARACTER_STORAGE_SIZE est la valeur en bits d'une donnée déclarée de type CHARACTER (8 généralement).

Remarques

- toute unité de programme devra comporter l'instruction « use ISO_FORTRAN_ENV » pour accéder à ces constantes ;
- dans le cas où la taille demandée n'est pas disponible pour le compilateur utilisé, la constante correspondante renfermera la valeur -2 si une taille supérieure est disponible, -1 sinon.

Exemple

```
use ISO_FORTRAN_ENV
real(kind=REAL64)    :: x ! réel sur 64 bits.
integer(kind=INT64) :: i ! entier sur 64 bits.
```


On a la possibilité d'indiquer le **sous-type** désiré lors de l'écriture des constantes.

Il suffira, pour cela, de les suffixer (pour les constantes numériques) ou de les préfixer (pour les constantes chaînes de caractères) par la valeur du **sous-type** voulu en utilisant le caractère « _ » comme séparateur.

Exemple

```
23564_4                  ! INTEGER(KIND=4)
12.879765433245_8        ! REAL(KIND=8)
12.879765433245_REAL64   ! REAL(KIND=REAL64) (depuis la norme 2008)
1_"wolfy"                ! CHARACTER(LEN=5,KIND=1)
2_"wolfy"                ! CHARACTER(LEN=5,KIND=2)
integer(kind=short), parameter :: kanji = 2
kanji_"wolfy"            ! CHARACTER(LEN=5,KIND=kanji)
```

Fonction KIND

Cette fonction renvoie une valeur entière qui correspond au **sous-type** de l'argument spécifié.

Exemples

```
! Valeur associée au sous-type réel simple précision :
kind(1.0)

! Valeur associée au sous-type réel double précision :
kind(1.0d0)

! L'instruction suivante permet de déclarer un réel
! double précision quelle que soit la machine utilisée :
real(kind=kind(1.d0))
```

Exemple

```
program p
  double precision d
  real r

  r = 2.718
  call sp(real(r, kind(d)), d)
      ...
end program p
!-----------------------------------
subroutine sp(x, y)
  double precision x, y

  y = x*x
end subroutine sp
```

Remarque *: les types définis via cette fonction KIND sont assurés d'être portables au niveau de la compilation.*
Les fonctions SELECTED_REAL_KIND et SELECTED_INT_KIND vues ci-après vont plus loin en assurant la portabilité au niveau de l'exécution (sauf impossibilité matérielle détectée à la compilation).

Fonctions intrinsèques SELECTED_INT_KIND, SELECTED_REAL_KIND

Elles ont pour prototype :

```
SELECTED_INT_KIND(r), SELECTED_REAL_KIND(p,r)
```

La première reçoit un nombre entier **r** en argument et retourne une valeur qui correspond au **sous-type** permettant de représenter les entiers **n** tels que :

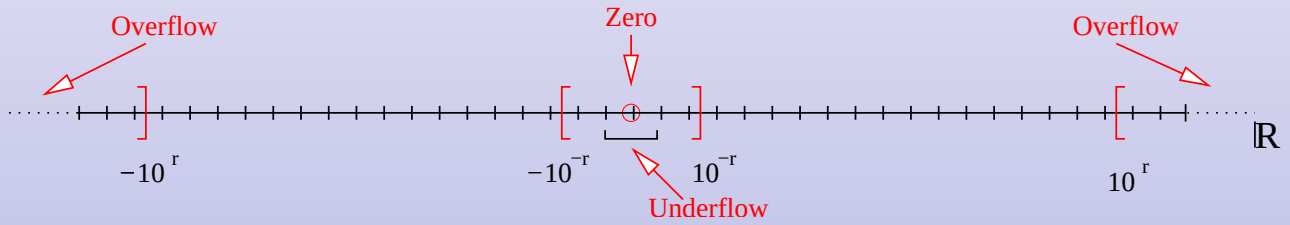
$$-10^r < n < 10^r$$

Elle retourne -1 si aucun **sous-type** ne répond à la demande.
La deuxième admet deux arguments **p** et **r** indiquant respectivement la **précision** (nombre de chiffres décimaux significatifs) et l'**étendue** (*range*) désirées. Elle retourne un entier (`kind`) qui correspond au **sous-type** permettant de représenter les **réels x** répondant à la demande avec :

$$10^{-r} <\mid x \mid< 10^r$$

Les arguments **p** et **r** sont optionnels, toutefois l'un des deux doit obligatoirement être fourni.
Cette fonction retourne -1 si la **précision** demandée n'est pas disponible, -2 si c'est l'**étendue** et -3 si ni l'une ni l'autre ne le sont.

Schéma de représentation des nombres réels pour une variante donnée :



Exemple

```
integer,parameter :: prec  =  selected_real_kind(p=9,r=50)
integer,parameter :: iprec =  selected_int_kind(r=2)

integer(kind=iprec)   :: k=1_iprec
real(kind=prec), save :: x
real(prec),      save :: y
x = 12.765_prec
   ...
selected_int_kind(30) ! Impossible -> -1
selected_real_kind(8)
selected_real_kind(9, 99)
selected_real_kind(r=50)
```

À noter que la **précision** et l'**étendue** peuvent être évaluées en utilisant les fonctions PRECISION et RANGE vues ci-après.

Fonctions intrinsèques RANGE et PRECISION

Pour le **sous-type** de l'argument *entier* ou *réel* fourni, la fonction RANGE retourne la valeur entière maximale de l'exposant décimal **r** telle que tout entier ou réel satisfaisant :

$$|entier| < 10^r$$

$$10^{-r} < |\text{réel}| < 10^r$$

est représentable.

La fonction PRECISION retourne la précision décimale (nombre maximum de chiffres significatifs décimaux — mantisse) pour le **sous-type** de l'argument réel fourni.

Exemples	Cray C90	Machines IEEE
range(1_4)	9	9
range(1_8)	18	18
range(1.0)	2465	37
precision(1.0)	13	6
range(1.d0)	2465	307
precision(1.d0)	28	15

Clauses RESULT/RECURSIVE

En **Fortran 90** on peut écrire des procédures (sous-programmes ou fonctions) récursives.

Définition :
- recursive function f(x) result(f_out)
- recursive subroutine sp(x, y, ...)
- recursive logical function f(n) result(f_out)
- logical recursive function f(n) result(f_out)

Attention : dans le cas d'une fonction récursive, pour que l'emploi du nom de la fonction dans le corps de celle-ci puisse indiquer un appel récursif, il est nécessaire de définir une variable résultat par l'intermédiaire de la clause `RESULT` lors de la définition de la fonction.

Remarques

- le type de la variable résultat est toujours celui de la fonction,
- possibilité d'utiliser la clause `RESULT` pour les fonctions non récursives.

Suite de Fibonacci

- $u_0 = 1$
- $u_1 = 1$
- $u_2 = 2$
- . . . .
- $u_n = u_{n-1} + u_{n-2}$

Exemple : suite de Fibonacci

```
recursive function fibonacci(n) result(fibo)
  integer, intent(in):: n
  integer            :: fibo
  integer, save      :: penult, antepenult
  !-------------------------------------------------
  if (n <= 1) then  !--> Test d'arrêt (On dépile)
    fibo = 1
    antepenult = 1 ; penult = 1
  !-------------------------------------------------
  else !--> Bloc récursif d'empilement dans la pile (stack)
    fibo = fibonacci(n-1); fibo = fibo + antepenult
    antepenult = penult ; penult = fibo
  end if
end function fibonacci
```

Attention, au niveau du bloc `ELSE` :

1. Il serait tentant de programmer cette fonction sous la forme :
   `fibo = fibonacci(n-1) + fibonacci(n-2)` qui est plus proche de la définition mathématique de la suite. Bien que parfaitement valide, cette dernière solution serait prohibitive en terme de performance car elle empilerait deux appels récursifs (au lieu d'un seul) et conduirait à recalculer de très nombreux termes déjà évalués !
2. Une autre possibilité serait de programmer sous la forme :
   `fibo = fibonacci(n-1) + antepenult` qui est une expression dont la valeur est indéterminée ; en effet, dans une telle expression l'appel de la fonction n'a pas le droit de modifier une entité (ici la variable locale `antepenult` avec l'attribut `SAVE`) intervenant également dans cette expression. Dans notre exemple, l'appel de la fonction `fibonacci` doit obligatoirement précéder le cumul de `antepenult` dans `fibo` d'où le découpage en deux instructions.

Note : pour un exemple de sous-programme récursif, cf. page 165

- **Tableau** : *objet* regroupant des données de même type repérées par un/des indices numériques.
- Nécessité de définir un objet composite (**structure de données**) regroupant des données (**champs** ou **composantes**) hétérogènes. Chaque champ est identifié par son nom. Sa déclaration nécessite la définition préalable du **type dérivé** étendant les types prédéfinis.

Exemple: manipuler des couleurs en composantes additives RVB

1. Définition du type dérivé `COULEUR` :

```
type COULEUR
  character(len=25)      :: nom
  integer, dimension(3) :: code_rvb
end type COULEUR
```

Norme 95 : possibilité d'initialisation des champs.

2. Déclaration du tableau `TABRVB` des 3 couleurs de base et initialisation :

```
type(COULEUR),dimension(3),parameter :: TABRVB = &
   (/ couleur("Rouge", (/ 255,0,0 /)),           &
      couleur("Vert ", (/ 0,255,0 /)),           &
      couleur("Bleu ", (/ 0,0,255 /)) /)
```

Initialisation (constructeur de structure)

Dans l'expression : `(/ couleur('Rouge', (/ 255,0,0 /) ),&...`

bien distinguer :

1. notion de *constructeur de structure* (*Structure Constructor*) :
   fonction (ici couleur) de même nom que le type dérivé ayant pour arguments les valeurs à placer dans les divers champs. Automatiquement créée, elle permet l'initialisation ou l'affectation globale d'une structure de données ;
2. notion de *constructeur de tableau* (*Array Constructor*) :
   agrégat vectoriel (séquence de valeurs scalaires sur une seule dimension) délimité par les caractères « **(/** » et « **/)** » (ou bien « **[** » et « **]** » depuis la norme 2003) permettant l'initialisation ou l'affectation globale d'un tableau de rang 1.

Exemple

```
integer,        dimension(3) :: TJ
type(couleur), dimension(5) :: TC

TC(1) = couleur( "Gris souris",      (/ 158, 158, 158 /) )
TC(2) = couleur( "Gris anthracite",  [ 48, 48, 48 ] )
TJ    = [ 255, 255, 0 ]
TC(3) = couleur( "Jaune",TJ )
```


Constructeur de structure : norme 2003

- depuis la **norme** 95 il est permis de préciser une valeur pour une composante lors de la définition du type ;
- lors de la valorisation d'un type dérivé via un *constructeur de structure*, la **norme** 2003 permet désormais d'affecter les composantes par mots clés ;
- si une composante d'un type dérivé à une valeur par défaut, l'argument correspondant au niveau du constructeur se comporte comme un argument optionnel.

Exemple

```
type couleur
  character(len=25)      :: nom = 'noir'
  integer, dimension(3) :: code_rvb = [ 0, 0, 0 ]
end type couleur
    ...
type(couleur) :: c
    ...
c = couleur( nom="Saumon", code_rvb=[ 248, 142, 85 ] )
    ...
c = couleur( code_rvb=[ 249, 66, 158 ], nom="Rose bonbon" )
    ...
c = couleur()
```

Constructeur de structure : norme 2003

Remarques

- pour une composante comportant l'attribut `pointer`, c'est une association (au sens du symbole « => ») qui s'effectuera avec l'argument précisé au niveau du constructeur. On pourra indiquer la fonction `NULL` pour forcer la composante à l'état nul de façon explicite ;
- pour une composante comportant l'attribut `allocatable`, l'argument précisé au niveau du constructeur devra être de même type et de même rang. S'il a l'attribut `allocatable`, il peut être ou non alloué. Comme pour une composante `pointer`, on pourra préciser la fonction `NULL` mais sans argument : la composante sera alors dans l'état "non alloué".

Symbole « % » d'accès à un champ

Le symbole « % » permet d'accéder à un champ d'une structure de donnée. Voici quelques exemples :

- `TC` ⇒ tableau de structures de données de type dérivé `COULEUR` ;
- `TC(2) et TABRVB(3)` ⇒ structures de type `COULEUR` ;
- `TC(1)%nom` ⇒ champ `nom` (`"Gris souris"`) de `TC(1)` ;
- `TC(1)%code_rvb` ⇒ tableau de 3 entiers contenant les composantes `RVB` de la teinte `"Gris souris"` ;
- `TC(2)%code_rvb(2)` ⇒ entier : composante verte du `"Gris anthracite"` ;
- `TC%code_rvb(2)` ⇒ tableau de 5 entiers : composantes vertes ;
- `TC%code_rvb` ⇒ `INCORRECT !!` car au moins une des deux entités encadrant le symbole `%` doit être un scalaire (rang nul) sachant qu'**une structure est considérée comme un scalaire**. Dans ce cas, `TC` et `code_rvb` sont des tableaux de rang 1.

Remarque

- dans le cas où la partie gauche du symbole `%` est un tableau, l'opérande de droite ne doit pas avoir l'attribut `pointer` ni `allocatable` !

Symbole « % » d'accès à un champ

Supposons que nous voulions stocker dans `TC(4)` la couleur jaune sous la forme (`Rouge + Vert`) ; il serait tentant de le faire de la façon suivante :

```
TC(4) = tabrvb(1) + tabrvb(2)
```

Cette instruction ne serait pas valide car si le symbole d'affectation (=) est bien surchargé par défaut pour s'appliquer automatiquement aux structures de données, il n'en est pas de même de l'opérateur d'addition (+). Comme nous le verrons plus loin, seule l'affectation ayant un sens par défaut, la surcharge éventuelle d'opérateurs pour s'appliquer à des opérandes de type dérivé est possible mais à la charge du programmeur. Voilà pourquoi nous devons opérer au niveau des composantes de ces structures en utilisant le symbole %.

Exemple de nouvelle définition (couleur jaune)

```
TC(4)= &
  couleur("jaune", (/ tabrvb(1)%code_rvb(1) + tabrvb(2)%code_rvb(1), &
                      tabrvb(1)%code_rvb(2) + tabrvb(2)%code_rvb(2), &
                      tabrvb(1)%code_rvb(3) + tabrvb(2)%code_rvb(3) /))
```

ou plus simplement :

```
TC(4)=couleur( "Jaune", tabrvb(1)%code_rvb + tabrvb(2)%code_rvb )
```


Types dérivés et procédures

Une structure de données peut être transmise en argument d'une procédure et une fonction peut retourner un résultat de type dérivé.
Si le type dérivé n'est pas « **visible** » (par *use association* depuis un module ou par *host association* depuis la procédure hôte), il doit être défini à la fois (situation à éviter) dans l'appelé et l'appelant. Les deux définitions doivent alors :

- posséder toutes les deux l'attribut `SEQUENCE` ⇒ stockage des champs avec même ordre et mêmes alignements en mémoire ;
- être identiques. Le nom du type et celui de la structure peuvent différer **mais pas** le nom et la nature des champs.

Types dérivés et procédures

Exemple 1

```
type(COULEUR) :: demi_teinte ! Obligatoire ici !
     ...
TC(5) = demi_teinte(TC(1))
     ...
function demi_teinte(col_in)
  implicit none
  !-----------------------------------------------
  type COLOR  !<--- au lieu de COULEUR
    SEQUENCE  !<--- ne pas oublier dans l'appelant
    character(len=25)     :: nom
    integer, dimension(3) :: code_rvb
  end type COLOR
  !-----------------------------------------------
  type(COLOR) :: col_in, demi_teinte

  demi_teinte%nom      = trim(col_in%nom)//"_demi"
  demi_teinte%code_rvb = col_in%code_rvb/2
end function demi_teinte
```


Exemple 2

```
program geom3d
  implicit none
  integer i
  real    ps
  type VECTEUR
    real x,y,z
  end type VECTEUR
  type CHAMP_VECTEURS ! >>>> Types imbriqués
    integer n                              ! Nb. de vecteurs
    type(VECTEUR),dimension(20) :: vect ! 20 : taille max.
  end type CHAMP_VECTEURS
!-------------------------Déclarations --------
  type(VECTEUR)         :: u,v,w
  type(CHAMP_VECTEURS) :: champ
!-----------------------------------------------
  u=vecteur(1.,0.,0.)  !>>> Construct. struct.
  w=u                  !>>> Affectation
! champ=u              !>>> ERREUR
! if(u==v) then        !>>> ERREUR
       ...
  ps=prod_sca(u,v)
  champ%n=20 ; champ%vect=[ u, v, (w,i=1,champ%n-2) ]
contains
  function prod_sca(a,b)
    type(VECTEUR)      :: a,b
    real               :: prod_sca
    prod_sca=a%x*b%x + a%y*b%y + a%z*b%z
  end function prod_sca
end program geom3d
```

Types dérivés et entrées/sorties

Les **entrées/sorties** portant sur des objets de type dérivé (**ne contenant pas de composante pointeur**) sont possibles :

- avec format, les composantes doivent se présenter dans l'ordre de la définition du type et c'est portable ;
- sans format, la représentation binaire dans l'enregistrement est *constructeur dépendante* même avec SEQUENCE ; **non portable** !

Exemple

```
type couleur
  character(len=25)      :: nom = 'Noir'
  integer, dimension(3) :: code_rvb = [ 0, 0, 0 ]
end type couleur
!-----------------------------------------------
integer                          :: long
type(couleur), dimension(5) :: tc
    ...
inquire(iolength=long) tc; print *, "tc=", tc, "long=", long
    ...
write(unit=10, fmt="(5(A,3I4),I4)") tc, long
write(unit=11) tc, long
```

Remarques

- chaque champ peut être constitué d'éléments de type intrinsèque (`real`, `integer`, `logical`, `character`, etc.) ou d'un autre type dérivé imbriqué ;
- l'attribut PARAMETER est interdit au niveau d'un champ ;
- l'initialisation d'un champ à la définition du type n'est possible que depuis la norme 95. Un objet d'un tel type, à moins qu'il soit initialisé à sa déclaration, ne reçoit pas implicitement l'attribut SAVE. Il ne peut pas figurer dans un COMMON ;
- les normes 90/95 interdit l'attribut ALLOCATABLE au niveau d'un champ (valide depuis la norme 2003) ;
- un *objet* de type dérivé est considéré comme un **scalaire** mais :
  - un champ peut avoir l'attribut DIMENSION ;
  - on peut construire des tableaux de structures de données.
- l'attribut SEQUENCE pour un type dérivé est obligatoire si une structure de ce type :
  - est passée en argument d'une procédure externe au sein de laquelle une redéfinition du type est nécessaire ;
  - fait partie d'un COMMON.
- un champ peut avoir l'attribut POINTER **mais pas** TARGET.

Remarques

L'attribut pointer appliqué au champ d'une structure permet :

- la déclaration de *tableaux* de pointeurs via un tableau de structures contenant un champ unique ayant l'attribut `pointer` ; cf. paragraphe « *Tableaux de pointeurs* » du chapitre 7 page 159 ;
- la gestion de *listes chaînées* basées sur des types dérivés tels :

```
type cell
  real,dimension(4)   :: x
  character(len=10)   :: str
  type(cell),pointer  :: p
end type cell
```

  cf. l'exemple du chap. 7 page 165 et le corrigé de l'exercice 12 en annexe B ;
- l'allocation dynamique de mémoire appliquée à un champ de structure.

À noter : lors de l'affectation entre 2 structures (de même type), le compilateur réalise effectivement des affectations entre les composantes. Pour celles ayant l'attribut `pointer` cela revient à réaliser une association.

Introduction

Structure habituelle d'un programme en blocs :

```
[étiq:] IF (expression logique) THEN
            bloc1
        [ ELSE IF (expression logique) THEN [étiq]
            bloc2 ]
        [ ELSE [étiq]
            bloc3 ]
        END IF [étiq]
```

Note :

- l'étiquette (*if-construct-name*) `[étiq:]` optionnelle peut être utile pour clarifier des imbrications complexes de tels blocs.

Boucles DO

Forme générale

```
[étiquette:] DO [contrôle de boucle]
                  bloc
             END DO [étiquette]
```

1re forme

```
[étiquette:] DO variable = expr1, expr2[,expr3]
                  bloc
             END DO [étiquette]
```

Nombre d'itérations : $max\left(\frac{expr_2 - expr_1 + expr_3}{expr_3}, 0\right)$

Exemple

```
DO I=1,N
  C(I) = SUM(A(I,:)*B(:,I))
END DO
```


Boucles DO

2e forme

```
             DO WHILE (condition)
                  bloc
             END DO
```

Exemple

```
read(unit=11,iostat=eof)a, b, c
DO WHILE(eof == 0)
        . . . .
        . . . .
  read(unit=11, iostat=eof)a, b, c
END DO
```

Boucles DO

La dernière et **3e forme** sont les boucles DO sans contrôle de boucle :

- ⇒ instruction conditionnelle avec instruction EXIT dans le corps de la boucle pour en sortir.

```
DO
   séquence 1
   IF (condition ) EXIT
   séquence 2
END DO
```

Exemple

```
do
  read(*, *) nombre
  if (nombre == 0) EXIT
  somme = somme + nombre
end do
```

Remarques

- cette forme de boucle (événementielle) ne favorise guère l'optimiseur ;
- suivant que le test de sortie est fait en début ou en fin, cette boucle s'apparente au DO WHILE ou au DO UNTIL.

Boucles DO; bouclage anticipé ⇒ instruction CYCLE

Elle permet d'abandonner le traitement de l'itération courante et de passer à l'itération suivante.

Exemple

```
do
  read(*, *, iostat=eof)x
  if (eof /= 0) EXIT
  if (x <= 0.) CYCLE
  y = log(x)
end do
```

Note

- comme nous allons le voir ci-après, les instructions EXIT et CYCLE peuvent être étiquetées pour s'appliquer à la boucle portant l'étiquette (*do-construct-name*) spécifiée.

Boucles DO; instruction EXIT

Les instructions EXIT et CYCLE dans des boucles imbriquées nécessitent l'emploi de boucles étiquetées.

Exemple 1

```
implicit none
integer                :: i, l, n, m
real, dimension(10)    :: tab
real                   :: som, som_max
real                   :: res
som = 0.0
som_max=1382.
EXTER:&
do l = 1,n
  read *, m, tab(1:m)
  do i = 1, m
    call calcul(tab(i), res)
    if (res < 0.) CYCLE
    som = som + res
    if (som > som_max) EXIT EXTER
  end do
end do EXTER
```


Exemple 2

```
B1:&
do i = 1,n
  do j = 1, m
    call sp(i+j, r)
    if (r < 0.) CYCLE B1
  end do
end do B1
```

L'instruction SELECT CASE permet des branchements multiples qui dépendent de la valeur d'une expression scalaire de type entier, logique ou chaîne de caractères.

```
[ nom_bloc: ] SELECT CASE(expression)
              [ CASE(liste) [ nom_bloc ]
                   bloc_1]
                     ...
              [ CASE DEFAULT[ nom_bloc ]
                   bloc_n]
              END SELECT[ nom_bloc ]
```

- nom_bloc est une étiquette,
- expression est une expression de type INTEGER, LOGICAL ou CHARACTER,
- liste est une liste de constantes du même type que expression,
- bloc$_i$ est une suite d'instructions **Fortran**.

Exemple

```
PROGRAM structure_case
  integer :: mois, nb_jours
  logical :: annee_bissext
             ...
  SELECT CASE(mois)
    CASE(4, 6, 9, 11)
      nb_jours = 30
    CASE(1, 3, 5, 7:8, 10, 12)
      nb_jours = 31
    CASE(2)
    !-------------------------------------
       fevrier: select case(annee_bissext)
          case(.true.)
            nb_jours = 29
          case(.false.)
            nb_jours = 28
       end select fevrier
    !-------------------------------------
    CASE DEFAULT
      print *, " Numéro de mois invalide"
  END SELECT
END PROGRAM structure_case
```

structure block[1] : norme 2008

La norme 2008 de Fortran permet la construction de blocs au sein desquels il est permis d'effectuer de nouvelles déclarations. Les identificateurs ne sont visibles qu'à l'intérieur du bloc où ils sont déclarés et de ceux éventuellement imbriqués (à l'exception des blocs dans lesquels ces identificateurs font l'objet d'une redéclaration).

Exemple

```
PROGRAM structure_block
  integer ios
      ...
  do
    read( UNIT=1, ..., IOSTAT=ios ) ...
    if ( ios < 0 ) exit
    if ( ios > 0 ) then
      block
        character(len=100) :: message

        write(message, '(a,i0)') &
           "La lecture s'est interrompue &
          & suite à une erreur de numéro ", ios
        stop trim(message)
      end block
    end if
            ...
  end do
END PROGRAM structure_block
```

1. Cette nouveauté est disponible avec les compilateurs « gfortran » du GNU, « xlf » d'IBM et « ifort » d'INTEL (à partir de la version 15)

structure block : norme 2008

La structure block peut être étiquetée comme les structures précédentes. L'instruction EXIT référençant une telle structure peut rendre la programmation plus aisée.

Exemple

```
PROGRAM structure_block
       ...
  Etiq: block
    do
      if ( condition1 ) EXIT
      if ( condition2 ) EXIT Etiq
           ...
    end do
    call sp( ... )
  end block Etiq
END PROGRAM structure_block
```

Dans cet exemple on remarque que si la condition « condition2 » est réalisée, l'instruction « EXIT Etiq » permet de sortir de la boucle mais de plus d'éviter l'appel à la procédure sp.

Un tableau est un ensemble d'éléments du même type.
Pour déclarer un tableau, il suffit de préciser l'attribut `DIMENSION` lors de sa déclaration :

Exemple

```
integer, dimension(5)            :: tab
real(8), dimension(3,4)          :: mat
real,    dimension(-1:3,2,0:5)   :: a
```

Un tableau peut avoir jusqu'à 7 dimensions au maximum.

- Le **rang** (*rank*) d'un tableau est son nombre de dimensions.
- Le nombre d'éléments dans une dimension s'appelle l'**étendue** (*extent*) du tableau dans cette dimension.
- Le **profil** (*shape*) d'un tableau est un **vecteur** dont chaque élément est l'**étendue** du tableau dans la dimension correspondante.
- La **taille** (*size*) d'un tableau est le produit des éléments du vecteur correspondant à son **profil**.

Deux tableaux seront dits **conformants** s'ils ont **même profil**.
Attention : deux tableaux peuvent avoir la même taille mais avoir des profils différents ; si c'est le cas, ils ne sont pas conformants !

Exemple

```
real, dimension(-5:4,0:2) :: x
real, dimension(0:9,-1:1) :: y
real, dimension(2,3,0:5)  :: z
```

- Les tableaux `x` et `y` sont de **rang** 2, tandis que le tableau `z` est de **rang** 3 ;
- L'**étendue** des tableaux `x` et `y` est 10 dans la 1re dimension et 3 dans la 2e. Ils ont même **profil** : le vecteur `(/ 10, 3 /)`, ils sont donc **conformants**. Leur **taille** est égale à 30 ;
- Le **profil** du tableau `z` est le vecteur `(/ 2, 3, 6 /)`, sa **taille** est égale à 36.

Manipulations de tableaux (conformance, constructeur, section, taille, ...)

Fortran 90 permet de manipuler globalement l'ensemble des éléments d'un tableau.
On pourra, de ce fait, utiliser le nom d'un tableau dans des expressions. En fait, plusieurs opérateurs ont été **sur-définis** afin d'accepter des objets de type tableau comme opérande.
Il sera nécessaire, toutefois, que les tableaux intervenant dans une expression soient **conformants**.

Exemple

```
real,    dimension(6,7)       :: a,b
real,    dimension(2:7,5:11)  :: c
integer, dimension(4,3)       :: m
logical, dimension(-2:3,0:6)  :: l

m = 1    ! Tous les éléments valorisés à 1.
         ! Un scalaire est conformant à tout tableau.
b = 1.5  ! idem.
c = b
a = b + c + 4.
l = c == b
```


On notera que pour manipuler un tableau globalement, on peut soit indiquer son nom, comme dans les exemples précédents, soit indiquer son nom suivi entre parenthèses d'autant de caractères « : », séparés par des virgules, qu'il a de dimensions.

Reprise de l'exemple précédent

```
real,    dimension(6,7)       :: a,b
real,    dimension(2:7,5:11)  :: c
integer, dimension(4,3)       :: m
logical, dimension(-2:3,0:6)  :: l

m(:,:) = 1
b(:,:) = 1.5
c(:,:) = b(:,:)
a(:,:) = b(:,:) + c(:,:) + 4.
l(:,:) = c(:,:) == b(:,:)
```

On préfèrera la dernière notation à la précédente car elle a l'avantage de la clarté.

Initialisation de tableaux

Il est permis d'initialiser un tableau au moment de sa déclaration ou lors d'une instruction d'affectation au moyen de **constructeur de tableaux**.
Ceci n'est toutefois possible que pour les tableaux de **rang** 1. Pour les tableaux de **rang** supérieur à 1 on utilisera la fonction `reshape` que l'on détaillera plus loin.
Un **constructeur de tableau** est un vecteur de scalaires dont les valeurs sont encadrées par les caractères « **(/** » et « **/)** ».

Exemple

```
character(len=1), dimension(5) :: a = (/ "a", "b", "c", "d", "e" /)
integer,          dimension(4) :: t1, t2, t3
integer                        :: i
t1 = (/ 6, 5, 10, 1 /)
t2 = (/ (i*i, i=1,4) /)
t3 = (/ t2(1), t1(3), 1, 9 /)
```

Rappel : dans les « boucles implicites », il faut autant de « blocs parenthésés » (séparés par des virgules) qu'il y a d'indices : `(((i+j+k,i=1,3),j=1,4),k=8,24,2)`
À noter que chaque indice est défini par un triplet dont le troisième élément (optionnel) représente le pas.

Sections de tableaux

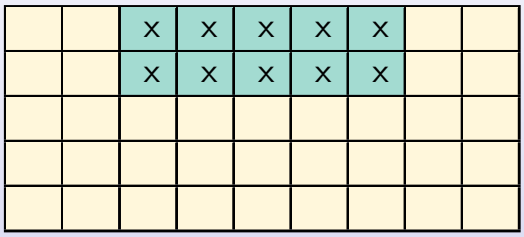
Il est possible de faire référence à une partie d'un tableau appelée **section de tableau** ou **sous-tableau**. Cette partie de tableau est également un tableau. De plus le tableau, dans son intégralité, est considéré comme le **tableau parent** de la partie définie. Le **rang** d'une **section de tableau** est inférieur ou égal à celui du **tableau parent**. Il sera inférieur d'autant d'indices qu'il y en a de fixés.
Il existe deux catégories de sections : les sections dites **régulières** et les sections **irrégulières**.
On désigne par **section régulière** un ensemble d'éléments dont les indices forment une progression arithmétique. Pour définir une telle **section** on utilise la **notation par triplet** de la forme « `val_init:val_fin:pas` » équivalent à une pseudo-boucle.
Par défaut, la valeur du `pas` est 1 et les valeurs de `val_init` et `val_fin` sont les limites définies au niveau de la déclaration du tableau parent. La notation *dégénérée* sous la forme d'un simple « **:** » correspond à l'étendue de la dimension considérée.

Exemple

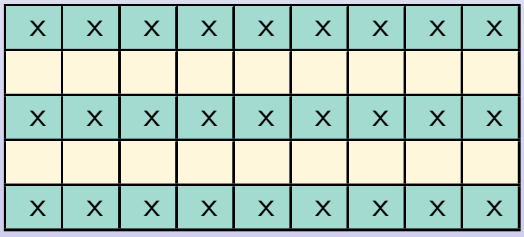
```
integer, dimension(10) :: a = (/ (i, i=1,10) /)
integer, dimension(6)  :: b
integer, dimension(3)  :: c

c(:)      = a(3:10:3)   ! <== "Gather"
b(1:6:2)  = c(:)        ! <== "Scatter"
```

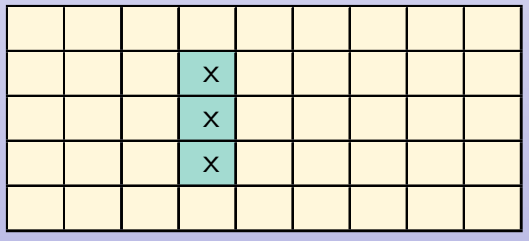
```
INTEGER, DIMENSION(5,9) :: T
```



← T(1:2,3:7) (rang=2, profil=(/ 2,5 /))



← T(1:5:2,:) (rang=2, profil=(/ 3,9 /))



← T(2:4,4) (rang=1, profil=(/ 3 /))

À noter :

T(i,j)	rang=0		Scalaire
T(i:i,j:j)	rang=2	profil=(/ 1,1 /)	Tableau dégénéré
T(2:,3)	rang=1	profil=(/ 4 /)	Vecteur


```
integer, dimension(10) :: a,b,c
integer, dimension(20) :: vec
integer                :: i

a(1:9:2)   = (/ (i,i=1,9,2) /)
a(2:10:2)  = (/ (i,i=-1,-9,-2) /)
b(:)       = (/ (i+1,i=1,7), a(1:3) /)
c(1:5)     = b(6:10)
c(6:10)    = a(1:5)
vec(4:13)  = a**2 + b**2 + c**2
```

Important : la valeur d'une expression tableau est **entièrement évaluée** avant d'être **affectée**.

Ainsi pour inverser un tableau on pourra écrire ⇒

```
real, dimension(20) :: tab
tab(:) = tab(20:1:-1)
```

Ce qui n'est pas du tout équivalent à ⇒

```
integer i
do i=1,20
  tab(i) = tab(21-i)
end do
```

Note :

- les *expressions tableaux* sont en fait des notations vectorielles ce qui facilite leur vectorisation puisque contrairement aux boucles, elles évitent au compilateur le contrôle des dépendances.

Sections irrégulières

Le triplet d'indices ne permet d'extraire qu'une séquence régulière d'indices. Il est possible d'accéder à des éléments quelconques par l'intermédiaire d'un **vecteur d'indices**. Il s'agit en fait d'une **indexation indirecte**.

Exemple

```
integer, dimension(10,9) :: tab
integer, dimension(3)    :: v_ind1
integer, dimension(4)    :: v_ind2

v_ind1 = (/ 2,7,6 /)
v_ind2 = (/ 4,1,1,3 /)
tab((/ 3,5,8 /), (/ 1,5,7 /)) = 1
```

- `tab(v_ind1,v_ind2)` est un **sous-tableau** à indices vectoriels. On remarque qu'il est constitué d'éléments répétés. Un tel **sous-tableau** ne peut pas figurer à gauche d'un signe d'affectation.
- `tab(v_ind2,5) = (/ 2,3,4,5 /)` n'est pas permis car cela reviendrait à vouloir affecter 2 valeurs différentes (3 et 4) à l'élément `tab(1,5)`.

Tableau en argument d'une procédure (taille et profil implicites)

Lorsque l'on passe un tableau en argument d'une procédure il est souvent pratique de pouvoir récupérer ses caractéristiques (*taille*, *profil*, ...) au sein de celle-ci.
En **Fortran 77** la solution était de transmettre, en plus du tableau, ses dimensions, ce qui est évidemment toujours possible en **Fortran 90**.

Exemple

```
integer, parameter       :: n=5,m=6
integer, dimension(n,m)  :: t=1
call sp(t,n,m)
end
subroutine sp(t,n,m)
integer                  :: n,m
integer, dimension(n,m)  :: t
print *,t
end
```

Si le tableau déclaré dans la procédure est de **rang r**, seules les **r-1** premières dimensions sont nécessaires car la dernière n'intervient pas dans le calcul d'adresses, d'où la possibilité de mettre un « * » à la place de la dernière dimension. À l'exécution de la procédure les **étendues** de chaque dimension, hormis la dernière, seront connues mais, la **taille** du tableau ne l'étant pas, c'est au développeur de s'assurer qu'il n'y a pas de débordement. Ce type de tableau est dit à « **taille implicite** » (*assumed-size-array*).

Reprise de l'exemple précédent

```
integer, parameter       :: n=5,m=6
integer, dimension(n,m)  :: t
t = 1
call sp(t,n)
end

subroutine sp(t,n)
  integer                    :: n
  integer, dimension(n,*)    :: t
  print *,size(t,1)
  print *,t(:,2)
  print *,t(1,:) ! Interdit
  print *,t      ! Interdit
end
```

La fonction **size** est une fonction intrinsèque qui retourne la **taille** du tableau passé en argument (sauf taille implicite).
Cette fonction admet un 2e argument optionnel qui permet de préciser la dimension suivant laquelle on désire connaître le nombre d'éléments. Dans ce cas, elle retourne en fait l'**étendue** relativement à la dimension spécifiée.
Dans l'exemple précédent, il est possible de récupérer l'**étendue** de toutes les dimensions du tableau **t** à l'exception de la dernière.

De plus, **Fortran 90** permet de transmettre le **profil** du tableau passé en argument. Pour cela, il faut que l'interface soit *explicite* (notion vue en détails plus loin au chapitre 8) ce qui permet de donner au compilateur un certain nombre de renseignements afin qu'il puisse mieux vérifier la cohérence entre arguments muets et arguments d'appel.

Exemple

```
program profil_implicite
  integer, dimension(5,6) :: t
  interface     !<<<<<<<<<<<<<<<<<<<<|
    subroutine sp(t)                  |
      integer, dimension(:,:) :: t    |
    end subroutine sp                 |
  end interface !<<<<<<<<<<<<<<<<<<<<|
  t = 0 ; call sp(t)
end program profil_implicite
subroutine sp(t)
  integer, dimension(:,:) :: t
  integer                 :: i
  print *,size(t,1), size(t,2)
  do i=1, size(t,1)
    print *,(t(i,j), j=1,size(t,2))
  end do
end subroutine sp
```

Un tableau passé de cette manière s'appelle un tableau à « **profil implicite** » (*assumed-shape-array*) ; dans ce cas, la cohérence du **type** et du **rang** est controlée.

Section de tableau non contiguë en argument d'une procédure

Une section de tableau peut être passée en argument d'une procédure.

Attention :
si elle constitue un ensemble de valeurs **non contiguës** en mémoire, le compilateur peut être amené à copier au préalable cette section dans un tableau d'éléments contigus passé à la procédure, puis en fin de traitement le recopier dans la section initiale...

⇒ **Dégradation possible des performances !**

En fait, cette copie (*copy in–copy out*) n'a pas lieu si les conditions suivantes sont réalisées :

- la section passée est régulière,
- l'argument muet correspondant est à profil implicite (ce qui nécessite que l'interface soit explicite).

C'est le cas de l'exemple suivant : le sous-programme sub1 reçoit en argument une section régulière non contiguë alors que le sous-programme sub2 reçoit le tableau dans sa totalité. Les temps d'exécutions sont analogues.

Exemple

```
program section
  implicit none
  integer, parameter             :: n = 1000, itermax=10000
  integer, parameter             :: i1 = 3, i2 = 70
  integer, parameter             :: j1 = 2, j2 = 50
  real, dimension(n,n)           :: a
  real, dimension(i1:i2,j1:j2) :: temp
  real c1, c2, c3
  integer i
  interface
    subroutine sub(x)
      real, dimension(:,:) :: x
    end subroutine sub
  end interface

  call random_number(a)

  call cpu_time(time=c1)
  do i=1,itermax
    call sub(a(i1:i2,j1:j2))
  end do
  call cpu_time(time=c2)
```

Exemple (suite)

```
  temp = a(i1:i2,j1:j2)
  do i=1,itermax
    call sub(temp)
  end do
  a(i1:i2,j1:j2) = temp
  call cpu_time(time=c3)

  print *, "Duree1 = ", c2-c1
  print *, "Duree2 = ", c3-c2
end program section

subroutine sub(x)
  real, dimension(:,:) :: x
  x = x*1.002
end subroutine sub
```

Sections non régulières en argument de procédures :

Si on passe une ***section non régulière*** *en argument d'appel d'une procédure, il faut savoir que :*

- *c'est une copie contiguë en mémoire qui est passée par le compilateur,*
- *l'argument muet correspondant de la procédure ne doit pas avoir la vocation* `INTENT(inout)` *ou* `INTENT(out)` *; autrement dit, en retour de la procédure, il n'y a pas mise-à-jour du tableau « père » de la section irrégulière.*

Fonctions intrinsèques d'interrogation (maxloc, lbound, shape, ...)

`SHAPE(source)`

retourne le **profil** du tableau passé en argument.

`SIZE(array[,dim])`

retourne la **taille** (ou l'étendue de la dimension indiquée via `dim`) du tableau passé en argument.

`UBOUND(array[,dim])`, `LBOUND(array[,dim])`

retournent les **bornes supérieures/inférieures** de chacune des dimensions (ou seulement de celle indiquée via `dim`) du tableau passé en argument.

```
integer, dimension(-2:27,0:49) :: t
```

⇒

SHAPE(t)	⇒	(/ 30,50 /)
SIZE(t)	⇒	1500
SIZE(t,dim=1)	⇒	30
SIZE(SHAPE(t))	⇒	2 (rang de t)
UBOUND(t)	⇒	(/ 27,49 /)
UBOUND(t(:,:))	⇒	(/ 30,50 /)
UBOUND(t,dim=2)	⇒	49
LBOUND(t)	⇒	(/ -2,0 /)
LBOUND(t(:,:))	⇒	(/ 1,1 /)
LBOUND(t,dim=1)	⇒	-2

`MAXLOC(array,dim[,mask])` ou `MAXLOC(array[,mask])`
`MINLOC(array,dim[,mask])` ou `MINLOC(array[,mask])`

retournent, pour le tableau `array` de rang **n** passé en argument, l'**emplacement de l'élément maximum/minimum** :

- de l'ensemble du tableau dans un tableau entier de rang 1 et de taille **n** si `dim` n'est pas spécifié,
- de chacun des vecteurs selon la dimension `dim` dans un tableau de rang **n-1** si `dim` est spécifié ; si **n=1**, la fonction retourne donc un scalaire.

`MASK` est un tableau de type `logical` conformant avec `array`.

`DIM=i` ⇒ la fonction travaille globalement sur cet indice (c.-à-d. un vecteur) pour chaque valeur fixée dans les autres dimensions.

Ainsi, pour un tableau `array` de rang 2, la fonction travaille sur :

- les vecteurs `array(:,j)` c.-à-d. les colonnes si `DIM=1`,
- les vecteurs `array(i,:)` c.-à-d. les lignes si `DIM=2`.

Exemple

```
MAXLOC( (/ 2,-1,10,3,-1 /) )          ⇒  (/ 3 /)
MINLOC( (/ 2,-1,10,3,-1 /),dim=1 )    ⇒  2
```

```
integer , dimension(0:2,-1:2) :: A
```

$$A=\begin{pmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & -4 \end{pmatrix} \Rightarrow$$

```
MAXLOC(array=A, mask=A.LT.5) ⇒ (/ 2,2 /)
MINLOC(array=A, mask=A.GT.5) ⇒ (/ 3,3 /)
MINLOC(array=A, mask=A>8 )   ⇒ [ 0,0 ]
MAXLOC(A, dim=2)             ⇒ (/ 3,2,3 /)
MAXLOC(A, dim=1)             ⇒ (/ 2,3,1,2 /)
MAXLOC(A, dim=2, mask=A<5)   ⇒ (/ 1,2,1 /)
MAXLOC(A, dim=1, mask=A<5)   ⇒ (/ 2,2,2,2 /)
```

Note :

- si **array** et **mask** sont de rang **n** et de profil (/ $d_1, d_2, \ldots, d_n$ /) et si dim=i est spécifié, le tableau retourné par ce type de fonctions sera de rang **n-1** et de profil (/ $d_1, d_2, \ldots, d_{i-1}, d_{i+1}, \ldots, d_n$ /)


```
FINDLOC(array,value,dim[,mask,kind,back]) ou
FINDLOC(array,value[,mask,kind,back])
```

retourne, pour le tableau ARRAY de rang n passé en argument, l'emplacement de l'élément fourni au moyen de l'argument VALUE. Le résultat est :

- un tableau entier de rang 1 et de taille n si DIM n'est pas spécifié : celui-ci contient les indices de l'élément recherché en traitant le tableau dans sa globalité ;
- un tableau de rang n-1 si DIM est spécifié : celui-ci contient les emplacements de l'élément recherché pour chacun des vecteurs selon la dimension DIM indiquée ;
- si MASK est spécifié, il devra être conformant avec le tableau ARRAY ;
- s'il existe plusieurs éléments correspondant à la valeur recherchée et si BACK est présent avec comme valeur .TRUE., alors c'est la dernière de ces valeurs selon l'ordre des éléments en mémoire qui est considérée.
- si le paramètre KIND est précisé, il indiquera la variante du type INTEGER désirée pour les valeurs retournées : en son absence, ces valeurs seront du type INTEGER par défaut ;
- si l'élément recherché n'existe pas ou bien si le tableau ARRAY est de taille nulle, alors le résultat sera constitué de 0.

Exemple

```
FINDLOC([2,6,4,6], VALUE=6)                ⇒ [2]
FINDLOC([2,6,4,6], VALUE=6, BACK=.TRUE.)   ⇒ [4]
FINDLOC([2,6,4,6], VALUE=6, DIM=1)         ⇒ 2
```

$$A = \begin{pmatrix} 0 & -5 & 7 & 7 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & 6 & 7 \end{pmatrix} ; B = \begin{pmatrix} 1 & 2 & -9 \\ 2 & 2 & 6 \end{pmatrix} ; M = \begin{pmatrix} T & T & F & T \\ T & T & F & T \\ T & T & F & T \end{pmatrix}$$

- `FINDLOC(ARRAY=A, VALUE=7, MASK=M)` ⇒ `[1,4]`
- `FINDLOC(ARRAY=A, VALUE=7, MASK=M, BACK=.TRUE.)` ⇒ `[3,4]`
- `FINDLOC(ARRAY=B, VALUE=2, DIM=1)` ⇒ `[2,1,0]`
- `FINDLOC(ARRAY=B, VALUE=2, DIM=2, BACK=.TRUE.)` ⇒ `[2,2]`

Fonctions intrinsèques de réduction

Selon que DIM est absent ou présent, toutes ces fonctions retournent soit un **scalaire** soit un tableau de rang **n-1** en désignant par **n** le rang du tableau passé en premier argument.

`ALL(mask[,dim])`

`DIM=i` ⇒ la fonction travaille globalement sur cet indice (c.-à-d. un vecteur) pour chaque valeur fixée dans les autres dimensions.

$$A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix} ; B = \begin{pmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{pmatrix} \Rightarrow A/=B = \begin{pmatrix} T & F & F \\ T & F & T \end{pmatrix}$$

- Réduction globale : `ALL(A.NE.B)` ⇒ `.false.`
- Réduction par colonne :
  `ALL(A.NE.B, dim=1)` ⇒ `(/ .true.,.false.,.false. /)`
- Réduction par ligne :
  `ALL(A.NE.B, dim=2)` ⇒ `(/ .false.,.false. /)`
- Comparaison globale de deux tableaux : `if (ALL(A==B)) ...`
- Test de conformance (entre tableaux de même rang) :
  `if (ALL(shape(A) == shape(B)))...`

`ANY(mask[,dim])`

$$A=\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix} ; B=\begin{pmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{pmatrix} \Rightarrow A/=B = \begin{pmatrix} T & F & F \\ T & F & T \end{pmatrix}$$

- Réduction globale : `ANY(A/=B)` ⇒ `.true.`
- Réduction par colonne :
  `ANY(A/=B, dim=1)` ⇒ `(/ .true.,.false.,.true. /)`
- Réduction par ligne :
  `ANY(A/=B, dim=2)` ⇒ `(/ .true.,.true. /)`
- Comparaison globale de deux tableaux : `if (ANY(A/=B)) ...`
- Test de non conformance (entre tableaux de même rang) :
  `if (ANY(shape(A) /= shape(B)))...`

`COUNT(mask[,dim])`

1. `COUNT( (/ .true.,.false.,.true. /) )` ⇒ 2
2. $A=\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix} ; B=\begin{pmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{pmatrix} \Rightarrow A/=B = \begin{pmatrix} T & F & F \\ T & F & T \end{pmatrix}$

   - Décompte global des valeurs vraies :
     `COUNT(A/=B)` ⇒ `3`
   - Décompte par colonne des valeurs vraies :
     `COUNT(A/=B,dim=1)` ⇒ `(/ 2,0,1 /)`
   - Décompte par ligne des valeurs vraies :
     `COUNT(A/=B,dim=2)` ⇒ `(/ 1,2 /)`

```
MAXVAL(array,dim[,mask]) ; MAXVAL(array[,mask])
MINVAL(array,dim[,mask]) ; MINVAL(array[,mask])
```

1. `MINVAL( (/ 1,4,9 /) )` ⇒ `1`, `MAXVAL( (/ 1,4,9 /) )` ⇒ `9`

2. $A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$ ⇒

`MINVAL(A,dim=1)`	⇒	`(/ 1,3,5 /)`
`MINVAL(A,dim=2)`	⇒	`(/ 1,2 /)`
`MAXVAL(A,dim=1)`	⇒	`(/ 2,4,6 /)`
`MAXVAL(A,dim=2)`	⇒	`(/ 5,6 /)`
`MINVAL(A,dim=1,mask=A>1)`	⇒	`(/ 2,3,5 /)`
`MINVAL(A,dim=2,mask=A>3)`	⇒	`(/ 5,4 /)`
`MAXVAL(A,dim=1,mask=A<6)`	⇒	`(/ 2,4,5 /)`
`MAXVAL(A,dim=2,mask=A<3)`	⇒	`(/ 1,2 /)`

Note :

- si le masque est partout faux, `MINVAL` retourne la plus grande valeur représentable (dans le type associé à `A`) et `MAXVAL` la plus petite.


```
PRODUCT(array,dim[,mask]) ; PRODUCT(array[,mask])
SUM(array,dim[,mask])     ; SUM(array[,mask])
```

1. `PRODUCT( (/ 2,5,-6 /) )` ⇒ `-60`, `SUM( (/ 2,5,-6 /) )` ⇒ `1`

2. $A = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$ ⇒

`PRODUCT(A,dim=1)`	⇒	`(/ 2,12,30 /)`
`PRODUCT(A,dim=2)`	⇒	`(/ 15,48 /)`
`SUM(A,dim=1)`	⇒	`(/ 3,7,11 /)`
`SUM(A,dim=2)`	⇒	`(/ 9,12 /)`
`PRODUCT(A,dim=1,mask=A>4)`	⇒	`(/ 1,1,30 /)`
`PRODUCT(A,dim=2,mask=A>3)`	⇒	`(/ 5,24 /)`
`SUM(A,dim=1,mask=A>5)`	⇒	`(/ 0,0,6 /)`
`SUM(A,dim=2,mask=A<2)`	⇒	`(/ 1,0 /)`

Note :

- si le masque est partout faux, ces fonctions retournent l'élément neutre de l'opération concernée.

Fonctions intrinsèques de multiplication

```
DOT_PRODUCT(vector_a,vector_b)
MATMUL(matrix_a,matrix_b)
```

DOT_PRODUCT retourne le produit scalaire des deux vecteurs passés en argument,
MATMUL effectue le produit matriciel de deux matrices ou d'une matrice et d'un vecteur passés en argument.

1. `DOT_PRODUCT( (/ 2,-3,-1 /), (/ 6,3,3 /) ) = 0`

2. $A = \begin{pmatrix} 3 & -6 & -1 \\ 2 & 3 & 1 \\ -1 & -2 & 4 \end{pmatrix}$; $V = \begin{pmatrix} 2 \\ -4 \\ 1 \end{pmatrix} \Rightarrow$ MATMUL(A,V) $= \begin{pmatrix} 29 \\ -7 \\ 10 \end{pmatrix}$

Les deux fonctions DOT_PRODUCT et MATMUL admettent des vecteurs et/ou matrices de type logique en argument.

```
DOT_PRODUCT(v1,v2)
```

v1 de type entier ou réel	⇒	sum(v1*v2)
v1 de type complexe	⇒	sum(conjg(v1)*v2)
v1 et v2 de type logique	⇒	any(v1.and.v2)

```
c = MATMUL(a,b)
```

Si profil(a)=(/ n,p /) et profil(b)=(/ p,q /)	⇒ profil(c) = (/ n,q /) $c_{i,j}$=sum(a(i,:)*b(:,j))
Si profil(a)=(/ p /) et profil(b)=(/ p,q /)	⇒ profil(c) = (/ q /) c_j=sum(a*b(:,j))
Si profil(a)=(/ n,p /) et profil(b)=(/ p /)	⇒ profil(c) = (/ n /) c_i=sum(a(i,:)*b)
Si a et b de type logique	⇒ On remplace sum par any et * par .and. dans les formules précédentes

Fonctions intrinsèques de construction/transformation

`RESHAPE(source,shape[,pad][,order])`

Cette fonction permet de construire un tableau d'un **profil** donné à partir d'éléments d'un autre tableau.

1. `RESHAPE( (/ (i,i=1,6) /), (/ 2,3 /) )`

$$\Rightarrow \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

2. `RESHAPE( (/ ((i==j,i=1,4),j=1,3) /), (/ 4,4 /), &`
   `(/ .true., .true., .true., .true. /) )`

$$\Rightarrow \begin{pmatrix} T & F & F & T \\ F & T & F & T \\ F & F & T & T \\ F & F & F & T \end{pmatrix}$$

Notes :

- PAD : tableau de *padding* optionnel doit être *array valued* ; ça ne peut pas être un scalaire ;
- ORDER : vecteur optionnel contenant l'une des `n!` permutations de `(1,2,..,n)` où `n` représente le rang du tableau retourné ; il indique l'ordre de rangement des valeurs en sortie. Par défaut, pour `n=2`, ce vecteur vaut `(/ 1,2 /)` ; le remplissage se fait alors classiquement colonne après colonne car c'est l'indice 1 de ligne qui varie le plus vite. À l'inverse, le vecteur `(/ 2,1 /)` impliquerait un remplissage par lignes ;
- la fonction RESHAPE est utilisable au niveau des initialisations (cf. page 123).

Exemples

```
RESHAPE( source=(/ ((i==j,i=1,4),j=1,3) /), &
         shape=(/ 4,4 /),                   &
         pad=(/ (.true., i=1,4) /),         &
         order=(/ 2,1 /) )
```

$$\Rightarrow \begin{pmatrix} T & F & F & F \\ F & T & F & F \\ F & F & T & F \\ T & T & T & T \end{pmatrix}$$

`CSHIFT(array,shift[,dim])`

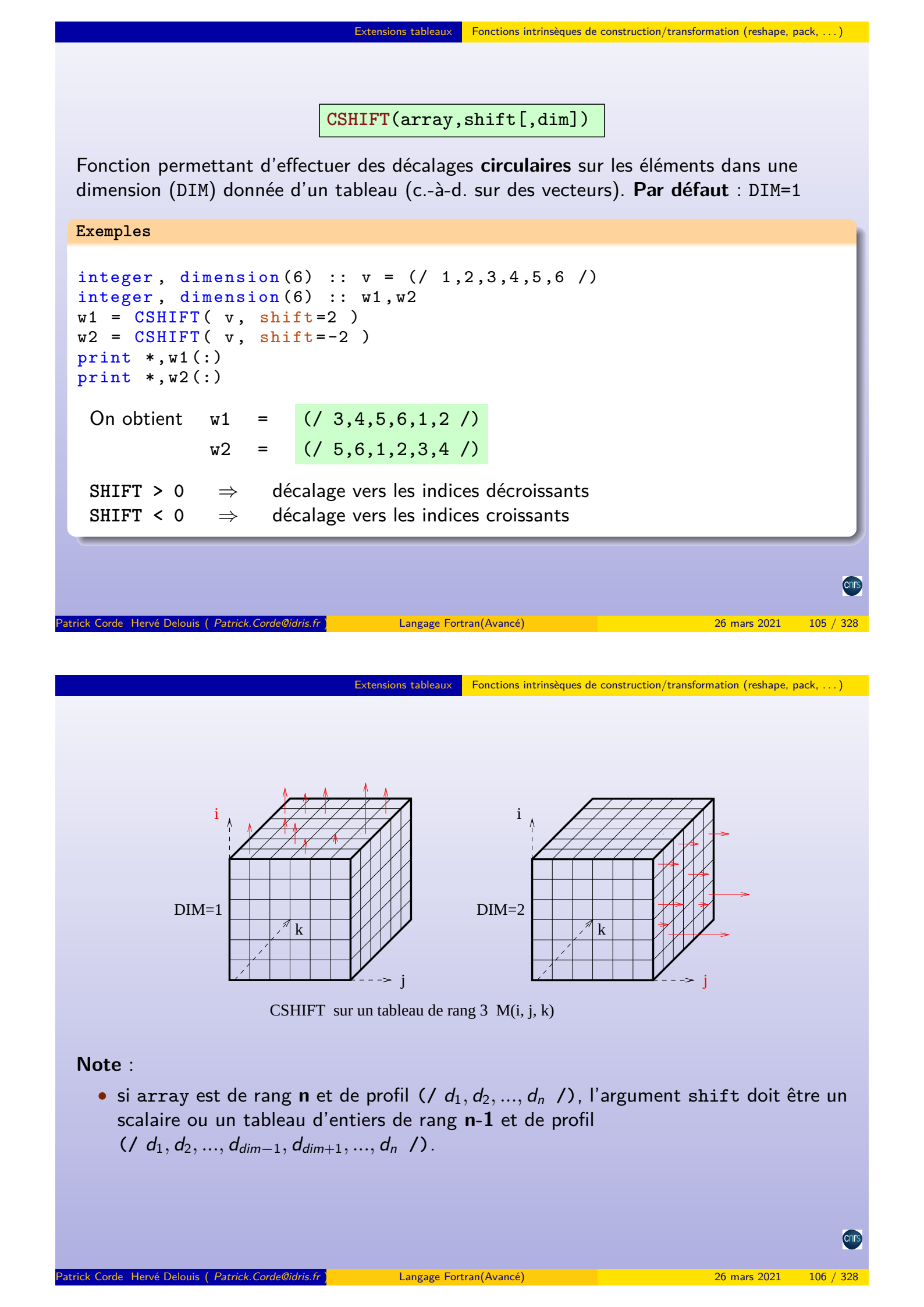
Fonction permettant d'effectuer des décalages **circulaires** sur les éléments dans une dimension (`DIM`) donnée d'un tableau (c.-à-d. sur des vecteurs). **Par défaut** : `DIM=1`

Exemples

```
integer, dimension(6) :: v = (/ 1,2,3,4,5,6 /)
integer, dimension(6) :: w1,w2
w1 = CSHIFT( v, shift=2 )
w2 = CSHIFT( v, shift=-2 )
print *,w1(:)
print *,w2(:)
```

On obtient `w1 = (/ 3,4,5,6,1,2 /)`
`w2 = (/ 5,6,1,2,3,4 /)`

`SHIFT > 0` $\Rightarrow$ décalage vers les indices décroissants
`SHIFT < 0` $\Rightarrow$ décalage vers les indices croissants



CSHIFT sur un tableau de rang 3 M(i, j, k)

Note :

- si `array` est de rang **n** et de profil `(/` $d_1, d_2, ..., d_n$ `/)`, l'argument `shift` doit être un scalaire ou un tableau d'entiers de rang **n-1** et de profil `(/` $d_1, d_2, ..., d_{dim-1}, d_{dim+1}, ..., d_n$ `/)`.

Soit **M** le tableau $\begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix}$

```
CSHIFT( array=M, shift=-1 )
```

$$\Rightarrow \begin{pmatrix} m & n & o & p \\ a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{pmatrix}$$

```
CSHIFT( array=M, shift=(/ 2,-2,-1,0 /), dim=2 )
```

$$\Rightarrow \begin{pmatrix} c & d & a & b \\ g & h & e & f \\ l & i & j & k \\ m & n & o & p \end{pmatrix}$$

Dérivée d'une matrice via CSHIFT

Soit à calculer la dérivée `D(M,N)` suivant la 2e dimension d'une matrice `F(M,N)` définie sur un domaine supposé cylindrique :

- **via une double boucle classique** :

```
do i=1,M
  do j=1,N
    D(i,j) = 0.5 * ( F(i,j+1) - F(i,j-1) )
  end do
end do
```

  Mais il faudrait rajouter le traitement périodique des données aux frontières du domaine cylindrique, c.-à-d. remplacer `F(:,N+1)` par `F(:,1)` et `F(:,0)` par `F(:,N)`

- **avec la fonction** `CSHIFT` :

```
D(:,:) = 0.5*( CSHIFT(F,1,2) - CSHIFT(F,-1,2) )
```

  La fonction `CSHIFT` traite automatiquement le problème des frontières.

`EOSHIFT(array,shift[,boundary][,dim])`

Cette fonction permet d'effectuer des décalages sur les éléments d'un tableau dans une dimension (`DIM`) donnée avec possibilité de remplacer ceux perdus (*End Off*) à la "frontière" (*boundary*) par des éléments de remplissage.

`SHIFT > 0` ⇒ décalage vers les indices décroissants

`SHIFT < 0` ⇒ décalage vers les indices croissants

Par défaut : `DIM=1`

Si, lors d'un remplacement, aucun élément de remplissage (`boundary`) n'est disponible celui par défaut est utilisé. Il est fonction du type des éléments du tableau traité.

Type du tableau	Valeur par défaut
`INTEGER`	0
`REAL`	0.0
`COMPLEX`	(0.0,0.0)
`LOGICAL`	.false.
`CHARACTER`(*len*)	*len* blancs

Exemples

```
integer, dimension(6) :: v = (/ (i,i=1,6) /)
integer, dimension(6) :: w1, w2
w1 = EOSHIFT( v, shift=3 )
w2 = EOSHIFT( v, shift=-2, boundary=100 )
```

On obtient : `w1 = (/ 4,5,6,0,0,0 /)`

`w2 = (/ 100,100,1,2,3,4 /)`

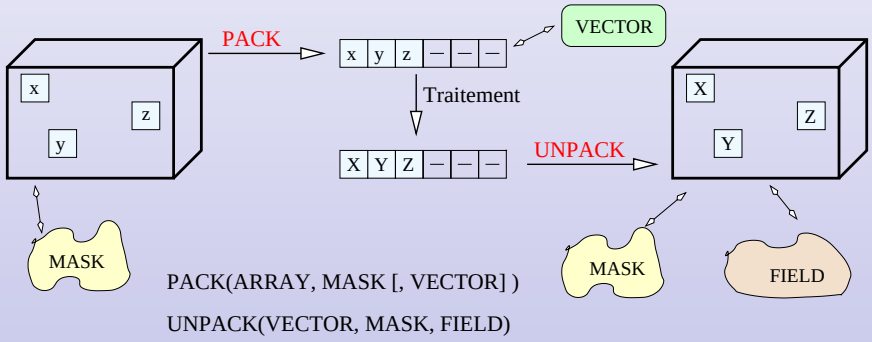
```
character, dimension(4,4) :: t1_car, t2_car

t1_car=RESHAPE( source = (/ (achar(i),i=97,112) /), &
                shape  = shape(t1_car) )
t2_car=EOSHIFT( array    = t1_car, &
                shift    = 3,      &
                boundary = (/ (achar(i),i=113,116) /) )
```

$$\Rightarrow \texttt{t1\_car} = \begin{pmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{pmatrix} ; \texttt{t2\_car} = \begin{pmatrix} d & h & l & p \\ q & r & s & t \\ q & r & s & t \\ q & r & s & t \end{pmatrix}$$

PACK(array,mask[,vector])

Fonction permettant de compresser un tableau sous le contrôle d'un masque. Le résultat est un vecteur.



Exemples

```
integer, dimension(3,3) :: a=0
a = EOSHIFT(a, shift=(/ 0,-1,1 /), boundary=(/ 1, 9,6 /))
print *, PACK(a, mask=a/=0)
print *, PACK(a, mask=a/=0, vector=(/ (i**3,i=1,5) /))
```

⇒ `(/ 9,6 /)` ; `(/ 9,6,27,64,125 /)`

Notes :

- pour linéariser une matrice : `PACK( a, mask=.true. )` ;
- à défaut de l'argument `VECTOR`, le résultat est un vecteur de taille égale au nombre d'éléments vrais du masque (`COUNT(MASK)`).
  Si `VECTOR` est présent, le vecteur résultat aura la même taille que lui (et sera complété en "piochant" dans `VECTOR`), ce qui peut être utile pour assurer la conformance d'une affectation. Le nombre d'éléments de `VECTOR` doit être égal ou supérieur au nombre d'éléments vrais du masque.

UNPACK(vector,mask,field)

Fonction décompressant un vecteur sous le contrôle d'un masque.
Pour tous les éléments vrais du masque, elle *pioche* les éléments de VECTOR et pour tous les éléments faux du masque, elle *pioche* les éléments correspondants de FIELD qui joue le rôle de « trame de fond ». MASK et FIELD doivent être conformants ; leur profil est celui du tableau retourné.

Exemples

```
integer, dimension(3)   :: v2 = 1
integer, dimension(3)   :: v = (/ 1,2,3 /)
integer, dimension(3,3) :: a, fld
logical, dimension(3,3) :: m
m = RESHAPE( source=(/ ((i==j,i=1,3),j=1,3) /),&
             shape = shape(m) )
fld = UNPACK( v2, mask=m, field=0 )
m   = CSHIFT( m,  shift=(/ -1,1,0 /), dim=2 )
a   = UNPACK( v,  mask=m, field=fld )
```

$$m \Rightarrow \begin{pmatrix} T & F & F \\ F & T & F \\ F & F & T \end{pmatrix} \quad fld \Rightarrow \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad m \Rightarrow \begin{pmatrix} F & T & F \\ T & F & F \\ F & F & T \end{pmatrix} \quad a \Rightarrow \begin{pmatrix} 1 & 2 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

Compression/décompression d'une matrice tridiagonale

```
integer,parameter             :: n=5
real,    dimension(n,n)       :: A
logical, dimension(n,n)       :: m
real,dimension(n + 2*(n-1))   :: v
!--Valorisation de la matrice A
. . . . . .
!--Création d'un masque tridiagonal
m=reshape( (/ ((i==j.or.i==j-1.or.i==j+1,i=1,n),j=1,n) /), &
           shape=shape(m) )
!--Compression (éléments tri-diagonaux)
v=pack( A,mask=m )
!--Traitement des éléments tri-diagonaux
!--compressés
v=v+1. ; . . . . .
!--Décompression après traitement
A=unpack( v,mask=m,field=A )
!--Impression
do i=1,size(A,1)
  print "(*(1x,f7.5))", A(i,:) ! « * » : fortran 2008
end do
```

SPREAD(source,dim,ncopies)

Duplication par ajout d'une dimension. Si **n** est le rang du tableau à dupliquer, le rang du tableau résultat sera **n+1**.

Exemples

1. **duplication selon les colonnes par un facteur 3**

```
integer, dimension(3) :: a = (/ 4,8,2 /)
print *, SPREAD(source=a, dim=2, ncopies=3)
```

$$\Rightarrow \begin{pmatrix} 4 & 4 & 4 \\ 8 & 8 & 8 \\ 2 & 2 & 2 \end{pmatrix}$$

2. **dilatation/expansion d'un vecteur par un facteur 4**

```
integer, dimension(3)  :: a = (/ 4,8,2 /)
print *, PACK( array=SPREAD(source=a, dim=1, ncopies=4), &
               mask=.true. )
```

$\Rightarrow$ `(/ 4 4 4 4 8 8 8 8 2 2 2 2 /)`

TRANSPOSE(matrix)

Cette fonction permet de transposer la matrice passée en argument.

Exemple

```
integer, dimension(3,3) :: a, b, c
a = RESHAPE( source = (/ (i**2,i=1,9) /), &
             shape  = shape(a) )
b = TRANSPOSE( matrix=a )
c = RESHAPE( source = a,          &
             shape  = shape(c), &
             order  = (/ 2,1 /) )
```

$\Rightarrow$ a = $\begin{pmatrix} 1 & 16 & 49 \\ 4 & 25 & 64 \\ 9 & 36 & 81 \end{pmatrix}$ b = $\begin{pmatrix} 1 & 4 & 9 \\ 16 & 25 & 36 \\ 49 & 64 & 81 \end{pmatrix}$ c = $\begin{pmatrix} 1 & 4 & 9 \\ 16 & 25 & 36 \\ 49 & 64 & 81 \end{pmatrix}$

La fonction suivante est une fonction élémentaire, c'est-à-dire que celle-ci a été définie pour traiter des scalaires et s'étend automatiquement à des arguments de type tableau.

`MERGE(tsource,fsource,mask)`

Exemple 1 : fusion de deux tableaux sous le contrôle d'un masque

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} ; b = \begin{pmatrix} -1 & -2 & -3 \\ -4 & -5 & -6 \\ -7 & -8 & -9 \end{pmatrix} ; m = \begin{pmatrix} T & F & F \\ F & T & F \\ F & F & T \end{pmatrix}$$

```
MERGE(tsource=a, fsource=b, mask=m)
```

$$\Rightarrow \begin{pmatrix} 1 & -2 & -3 \\ -4 & 5 & -6 \\ -7 & -8 & 9 \end{pmatrix}$$

Exemple 2 : en remplacement d'un test

```
use ISO_FORTRAN_ENV
real    x, y, z
logical flag

x = acos(-1.)
z = exp(1.)
read(INPUT_UNIT, *) flag

if (flag) then
  y = x
else
  y = z
end if

! autre écriture

y = MERGE(tsource=x, fsource=z, mask=flag)
```

Instruction et bloc WHERE

L'instruction WHERE permet d'effectuer des **affectations de type tableau** par l'intermédiaire d'un filtre (masque logique) :

```
[étiq:] WHERE (mask)
            bloc1
        ELSEWHERE [étiq]
            bloc2
        END WHERE [étiq]
```

Où mask est une expression logique retournant un tableau de logiques.
Remarque : *bloc1* et *bloc2* sont constitués uniquement d'instructions d'affectation portant sur des tableaux conformants avec le masque.

Exemple

```
real, dimension(10) :: a
     ...
WHERE (a > 0.)
  a = log(a)
ELSEWHERE
  a = 1.
END WHERE
```


Instruction et bloc WHERE

Ce qui est équivalent à :

```
do i = 1,10
   if (a(i) > 0.) then
      a(i) = log(a(i))
   else
      a(i) = 1.
   end if
end do
```

Remarques

- Lorsque *bloc2* est absent et que *bloc1* se résume à une seule instruction, on peut utiliser la forme simplifiée : `WHERE (expression_logique_tableau) instruction`

  Exemple ⇒ `WHERE(a>0.0) a = sqrt(a)`

- Dans l'exemple suivant : `WHERE(a>0.) a = a - sum(a)`, la fonction sum est évaluée comme la somme de tous les éléments de a, car sum n'est pas une fonction élémentaire (fonction que l'on peut appliquer séparément à tous les éléments d'un tableau).
  Cependant l'affectation n'est effectuée que pour les éléments positifs de a.

Instruction et bloc WHERE

- Considérons l'exemple suivant :

```
WHERE(a>0.) b = a/sum(sqrt(a))
```

  La règle veut que les fonctions élémentaires (ici **sqrt**) apparaissant en argument d'une fonction non élémentaire (ici **sum**) ne soient pas soumises au masque. L'expression sum(sqrt(a)) sera donc calculée sur tous les éléments de **a**. Cela provoquera bien sûr une erreur si l'une au moins des valeurs de **a** est négative.
- Lors de l'exécution d'une instruction ou d'un bloc WHERE le masque est évalué avant que les instructions d'affectation ne soient exécutées. Donc si celles-ci modifient la valeur du masque, cela n'aura aucune incidence sur le déroulement de l'instruction ou du bloc WHERE.
- On ne peut imbriquer des blocs WHERE.

Norme 95 : il est possible d'imbriquer des blocs WHERE qui peuvent être étiquetés de la même façon que le bloc select ou la boucle DO par exemple. De plus, le bloc WHERE peut contenir plusieurs clauses ELSEWHERE avec masque logique (sauf le dernier).

Expressions d'initialisation

Une expression d'initialisation (« *initialization-expression* ») doit être formée à l'aide de constantes ou d'expressions constantes. Ce type d'expression est utilisé lors d'une déclaration et peut être construite à l'aide d'éléments comme indiqués dans les exemples suivant :

- constructeur de vecteur (avec boucles implicites) :

```
integer i
integer, dimension(10) :: t1=(/ (i*2, i=1,10) /)
```

- constructeur de structure :

```
type(couleur) :: c2=couleur("Vert", (/ 0.,1.,0. /))
```

- fonctions intrinsèques élémentaires :

```
integer, parameter           :: n=12**4
integer(kind=2)              :: l=int(n, kind=2)
character(len=*), parameter  :: str = "CNRS/IDRIS"
integer                      :: k=index(str, "IDRIS")
real                         :: x=real(n)
```

- fonctions intrinsèques d'interrogation (LBOUND, UBOUND, SHAPE, SIZE, BIT_SIZE, KIND, LEN, DIGITS, EPSILON, HUGE, TINY, RANGE, RADIX, MAXEXPONENT, MINEXPONENT) :

```
real , dimension(10,20) :: a
integer                 :: d=size(a,1)*4
integer                 :: n=kind(0.D0)
```

- fonctions de transformation (REPEAT, RESHAPE, TRIM, SELECTED_INT_KIND, SELECTED_REAL_KIND, TRANSFER, NULL, ...) :

```
integer i
integer , dimension(4,2) :: t = &
     reshape( source=[ (i,i=1,8) ], shape=shape(t) )

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

Remarque

- La notation « [] » utilisée dans l'exemple précédent a été introduite par la norme 2003 pour l'écriture du constructeur de vecteur. Il est bien entendu que les caractères habituels « (/, /) » sont toujours valides.

Exemples d'expressions tableaux

$N!$	PRODUCT((/ (k,k=2,N) /))
$\sum_i a_i$	SUM(A)
$\sum_i a_i \cos x_i$	SUM(A*cos(X))
$\sum_{\lvert a_i\rvert<0.01} a_i \cos x_i$	SUM(A*cos(X), mask=ABS(A)<0.01)
$\sum_j \prod_i a_{ij}$	SUM(PRODUCT(A, dim=1))
$\prod_i \sum_j a_{ij}$	PRODUCT(SUM(A, dim=2))
$\sum_i (x_i - \overline{x})^2$	SUM((X - SUM(X)/SIZE(X))**2)
Linéarisation d'une matrice M	PACK(M, mask=.true.)
3eligne de M	M(3,:)
2ecolonne de M	M(:,2)
$\sum_{1\le i\le 3} \sum_{2\le j\le 4} M_{ij}$	SUM(M(1:3,2:4))

Exemples d'expressions tableaux

Trace : somme éléments diagonaux de M	`SUM(M, mask=reshape( & source=(/ ((i==j,i=1,n),j=1,n) /),& shape=shape(M)))`
Valeur max. matrice triangulaire inférieure de M	`MAXVAL(M, mask=reshape( & source=(/ ((i>=j,i=1,n),j=1,n) /),& shape=shape(M)))`
Dérivée selon les lignes (domaine cylindrique)	`( CSHIFT(M, shift= 1, dim=2) - & CSHIFT(M, shift=-1, dim=2) ) / 2.`
Décompte des éléments positifs	`COUNT(M > 0.)`
$\|M\|_1 = max_j \sum_i \|m_{ij}\|$	`MAXVAL(SUM(ABS(M), dim=1))`
$\|M\|_\infty = max_i \sum_j \|m_{ij}\|$	`MAXVAL(SUM(ABS(M), dim=2))`
Produit matriciel : $M1.M2^T$	`MATMUL(M1, TRANSPOSE(M2))`
Produit scalaire : $\vec{V}.\vec{W}$	`DOT_PRODUCT(V, W)`

Expressions de spécification

Une expression de spécification (« *specification-expression* ») est une expression formée au moyen de constantes et de variables dont les valeurs peuvent être déterminées à l'entrée d'une procédure avant toute exécution d'instructions exécutables. C'est une expression scalaire entière.
Ces variables peuvent être des arguments muets, définies dans un `COMMON` ou bien accessibles via « *use* ou *host-association* ».
Des fonctions intrinsèques telles que `LBOUND`, `UBOUND`, `SHAPE`, `SIZE`, `BIT_SIZE`, `KIND`, `LEN`, `DIGITS`, `EPSILON`, `HUGE`, `TINY`, `RANGE`, `RADIX`, `MAXEXPONENT`, `MINEXPONENT` peuvent apparaître au sein de ces expressions.
Ce type d'expression est employé pour préciser les dimensions de tableaux et les longueurs de chaînes de caractères comme dans l'exemple suivant :

Exemple

```
subroutine sp( n, m, c, ... )
  ! Déclarations des arguments
  integer n, m
  character(len=*) c
  ! Déclarations des variables locales
  real, dimension(n*m)   :: vec
  character(len=len(c)) :: chaine
        ...
end subroutine sp
```

Il est possible de définir au sein d'une procédure des tableaux dont la taille varie d'un appel à l'autre. Ils sont alloués dynamiquement à l'entrée de la procédure et libérés à sa sortie de façon implicite.
Pour cela ces tableaux seront dimensionnés à l'aide d'expressions de spécification renfermant des variables.
Ces tableaux sont appelés **tableaux automatiques** ; c'est le cas des tableaux `C` et `V` de l'exemple suivant :

Exemple

```
subroutine echange(a, b, taille)
 integer, dimension(:,:)                  :: a, b
 integer                                  :: taille
 integer, dimension(size(a,1),size(a,2))  :: C
 real,    dimension(taille)               :: V

 C = a
 a = b
 b = C
    ...
end subroutine echange
```

Remarques

- pour pouvoir exécuter ce sous-programme, l'interface doit être « explicite » (cf. chapitre 8),
- un tableau automatique ne peut être initialisé.

Tableaux dynamiques ALLOCATABLE, profil différé

Un apport intéressant de la norme Fortran 90 est la possibilité de faire de l'allocation dynamique de mémoire. Pour pouvoir allouer un tableau dynamiquement on spécifiera l'attribut `ALLOCATABLE` au moment de sa déclaration. Un tel tableau s'appelle tableau à **profil différé** (*deffered-shape-array*).
Son allocation s'effectuera grâce à l'instruction `ALLOCATE` à laquelle on indiquera le profil désiré. L'instruction `DEALLOCATE` permet de libérer l'espace mémoire alloué. De plus la fonction intrinsèque `ALLOCATED` permet d'interroger le système pour savoir si un tableau est alloué ou non.

Exemple

```
real, dimension(:,:), ALLOCATABLE :: a
integer                           :: n,m,etat

read *, n, m
if (.not. ALLOCATED(a)) then
  ALLOCATE(a(n,m),stat=etat)
  if (etat /= 0) then
    print *,"Erreur allocat. tableau a" ; stop 4
  end if
end if
    ...
DEALLOCATE(a)
```

Tableaux dynamiques ALLOCATABLE, profil différé : remarques

- attention : en cas de problème au moment de l'allocation et en l'absence du paramètre STAT=etat, l'exécution du programme s'arrête automatiquement avec un message d'erreur (traceback) ; s'il est présent, l'exécution continue en séquence et c'est à vous de tester la valeur retournée dans la variable entière etat qui est différente de zéro en cas de problème ;
- il n'est pas possible de réallouer un tableau déjà alloué. Il devra être libéré auparavant ;
- un tableau local alloué dynamiquement dans une unité de programme a un état indéterminé à la sortie (RETURN/END) de cette unité sauf dans les cas suivants :
  - l'attribut SAVE a été spécifié pour ce tableau ;
  - une autre unité de progr. encore active *a visibilité* par *use association* sur ce tableau déclaré dans un module ;
  - cette unité de progr. est interne. De ce fait (*host association*), l'unité hôte peut encore y accéder.

  Norme 95 : ceux restant à l'état indéterminé sont alors automatiquement libérés ;
- un tableau dynamique peut avoir l'attribut TARGET ; sa libération (*deallocate*) doit obligatoirement se faire en spécifiant ce tableau et en aucun cas un pointeur intermédiaire lui étant associé ;
- un tableau dynamique peut être transmis en argument. Selon les **normes 90/95** l'argument muet correspondant ne doit pas avoir l'attribut ALLOCATABLE : il devra être alloué avant l'appel et c'est un tableau classique qui sera alors récupéré dans la procédure appelée.

Argument muet ALLOCATABLE : norme 2003

- depuis la **Norme 2003**, il est possible de préciser l'attribut allocatable pour le retour d'une fonction ainsi que pour un argument muet d'une procédure : l'interface devra être explicite ;
- si l'argument muet a l'attribut allocatable, l'argument d'appel correspondant ne peut pas être précisé sous forme d'une section de tableau et doit aussi avoir cet attribut ainsi que le même rang et le même type/sous-type, sans être nécessairement déjà alloué : s'il l'est, il sera alors possible de récupérer ses bornes au sein de la procédure appelée (voir exemple ci-après).

Exemple

```
program alloca
  implicit none
  real, dimension(:), ALLOCATABLE :: vec_x, vec_y

  allocate( vec_y(-2:100) )
  call sp( vec_x, vec_y )
  print *, vec_x( (/ lbound(vec_x), ubound(vec_x) /) ), vec_y( (/ -2, 100 /) )
  deallocate( vec_x ); deallocate( vec_y )
CONTAINS
  subroutine sp( v_x, v_y )
    real, dimension(:), ALLOCATABLE, intent(inout) :: v_x, v_y
    ALLOCATE(v_x(256))
    call random_number(v_x); call random_number(v_y)
    print *, "Bornes inf/sup(v_y) : ", lbound( v_y ), ubound( v_y )
  end subroutine sp
end program alloca
```

Composante allouable d'un type dérivé : norme 2003

Désormais la **norme** 2003 autorise l'emploi de l'attribut **ALLOCATABLE** au niveau d'une composante d'un type dérivé. Les normes précedentes (90/95) n'autorisaient que l'attribut **POINTER** pour gérer des composantes dynamiques.

Exemple

```
program alloca
  type obj_mat
    integer :: N, M
    real, dimension(:,:), ALLOCATABLE :: A
  end type obj_mat
  type(obj_mat) :: mat

  read *, mat%N, mat%M
  allocate( mat%A(mat%N, mat%M) ); call random_number( mat%A )
  print *, mat%A( (/ 1, mat%N /), (/ 1, mat%M /) )
  deallocate( mat%A )
end program alloca
```

Remarques

- la composante A de mat est en vérité un descripteur opaque renfermant les caractéristiques du tableau allouable (adresse du tableau, profil, bornes inf/sup, ...) ;
- lorsque l'emplacement en mémoire d'un objet de type dérivé est libéré (explicitement via une instruction **deallocate** ou bien implicitement par le compilateur), si celui-ci admet des composantes avec l'attribut **ALLOCATABLE**, celles-ci seront automatiquement désallouées (voir exemple transparent suivant).

Libération automatique de zones dynamiques

Exemple

```
program zones_dyn
  implicit none
  type vecteur
    real x, y, z
  end type vecteur
  type champs_vecteurs
    integer n
    type(vecteur), dimension(:), allocatable :: champs
  end type champs_vecteurs
  type(champs_vecteurs), dimension(:), allocatable :: tab_champs
  integer nb_elts, i

  read *,nb_elts
  allocate(tab_champs(nb_elts))
  do i=1,nb_elts
    read *,tab_champs(i)%n
    allocate(tab_champs(i)%champs(tab_champs(i)%n))
        ...
  end do
    ...
  deallocate(tab_champs) ! Désallocation automatique des composantes « champs »
    ...
```

Allocation d'un scalaire ALLOCATABLE : norme 2003

L'attribut ALLOCATABLE peut dorénavant s'appliquer à un **scalaire**, notamment à une chaîne de caractères.

Exemple

```
program alloca_char
  implicit none
  character(len=256) :: ch_in
  character(len=:), ALLOCATABLE :: ch_out
  read( *, "(A)" ) ch_in; ch_out =[1] strdup( ch_in )
  print "(i0,1x,a)", len(ch_out), ch_out
  deallocate( ch_out )
contains
  function strdup( ch_in )
    character(len=*) :: ch_in
    character(len=:), ALLOCATABLE :: strdup
    allocate( character(len=len_trim(ch_in)) :: strdup ) ! Facultatif ici[1]
    strdup = trim( ch_in )
  end function strdup
end program alloca_char
```

Dans cet exemple, la fonction strdup retourne une chaîne de caractères allouable dont la longueur est celle de la chaîne passée en argument débarrassée des blancs de fin. L'allocation est effectuée à l'aide de l'instruction ALLOCATE en explicitant le type et la longueur désirée.

1. dans le cas où l'allocation n'est pas précisée, elle sera effectuée automatiquement lors de l'affectation (voir transparent suivant)

Allocation/réallocation via l'affectation : norme 2003

Une allocation/réallocation d'une entité ayant l'attribut ALLOCATABLE peut se faire implicitement lors d'une opération d'affectation du type :

`var_alloc = expression`

1. si var_alloc est déjà allouée, elle est automatiquement désallouée si des différences (concernant le profil par exemple) existent entre var_alloc et *expression* ;
2. si var_alloc est ou devient désallouée, alors elle est réallouée selon les caractéristiques de *expression*.

Voici un exemple permettant le traitement d'une chaîne de caractères de longueur variable :

```
character(:), ALLOCATABLE :: NAME
 . . . .
NAME = "Beethoven"; ...;  NAME = "Ludwig-Van Beethoven"
```

La variable scalaire NAME de type CHARACTER sera allouée lors de la première affectation avec une longueur LEN=9. Lors de la 2e affectation, elle sera désallouée puis réallouée avec une longueur LEN=20.

À noter que cette possibilité de réallocation dynamique facilite la gestion des chaînes dynamiques ainsi que le respect de la contrainte de conformance lors d'une affectation de tableaux. Ainsi par exemple :

```
real, ALLOCATABLE, dimension(:) :: x
     ...
!--allocate( x(count( masque )) ) <--- Devient inutile !
     ...
x = pack( tableau, masque )
```

Le tableau x est automatiquement alloué/réalloué avec le bon profil sans que l'on ait à se préoccuper du nombre d'éléments vrais de masque.

Note :

- ce processus d'allocation/réallocation automatique peut être inhibé en mentionnant explicitement tout ou partie de l'objet :

```
NAME(:)                 = "chaîne_de_caractères"
x(1:count( masque )) = pack( tableau, masque )
```

  À gauche de l'affectation, la présence du caractère « : » signifie qu'on fait référence à un sous-ensemble d'une entité (NAME ou x) qui doit exister et donc être déjà allouée ;
- avec le compilateur « *ifort* » d'INTEL il peut être nécessaire, suivant le niveau de version, de positionner l'option « *-assume realloc_lhs* » pour bénéficier de ce mécanisme d'allocation/réallocation automatique.

Procédure MOVE_ALLOC de réallocation : norme 2003

`MOVE_ALLOC(FROM, TO)`

Ce sous-programme permet de transférer une allocation d'un objet allouable à un autre dans le but , par exemple, d'étendre un objet déjà alloué.

- FROM fait référence à une entité allouable de n'importe quel type/rang ;
- TO fait référence à une entité allouable de type compatible avec FROM et de même rang ;

En retour de ce sous-programme, le tableau allouable TO désigne le tableau allouable FROM ; la zone mémoire préalablement désignée par TO est désallouée.
En fait, c'est un moyen permettant de *vider* le descripteur de FROM après l'avoir recopié dans celui de TO.

- si FROM n'est pas alloué en entrée, TO devient non alloué en sortie ;
- sinon, TO devient alloué avec les mêmes caractéristiques (type dynamique, paramètres de type, bornes de tableau, valeurs) que FROM avait en entrée ;
- si TO a l'attribut TARGET, tout pointeur initialement associé à FROM devient associé à TO.

Exemple

```
real, ALLOCATABLE, dimension(:) :: grid, tempgrid
      ...
ALLOCATE(grid(-N:N))           ! Allocation initiale de grid
      ...
ALLOCATE(tempgrid(-2*N:2*N)) ! Allocation d'une grille plus grande
      ...                      !
tempgrid(::2) = grid           ! Redistribution des valeurs de grid
      ...                      !
call MOVE_ALLOC( TO=grid, FROM=tempgrid )
```

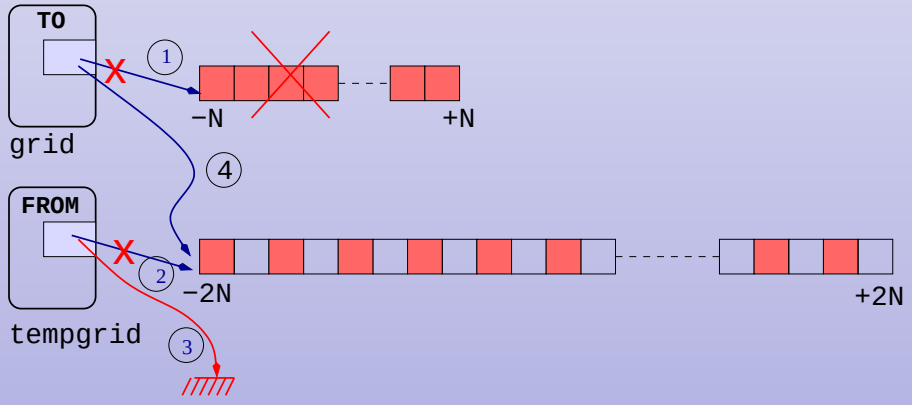


Figure 1 – *Schéma correspond au `move_alloc` précédent*

Exemple précédent en Fortran 90/95

```
real, ALLOCATABLE, dimension(:) :: grid, tempgrid
      ...                      !
ALLOCATE(grid(-N:N))           ! Allocation initiale de grid
      ...                      !
ALLOCATE(tempgrid(-2*N:2*N)) ! Allocation d'une grille plus grande
      ...                      !
tempgrid(::2) = grid           ! Redistribution des valeurs de grid
      ...                      !
DEALLOCATE( grid(:) )
ALLOCATE( grid(-2*N:2*N) )
grid(:) = tempgrid(:)
DEALLOCATE( tempgrid(:) )
```

Définition, états d'un pointeur

En **C, Pascal** ⇒ variable contenant l'adresse d'objets

En **Fortran 90** ⇒ alias d'objets

Tout pointeur Fortran a un état parmi les suivants :

1. **Indéfini** : à sa déclaration en tête de programme ;
2. **Nul** : alias d'aucun objet ;
3. **Associé** : alias d'un objet (cible).

Note *: pour ceux connaissant la notion de pointeur (type langage C), disons que le pointeur Fortran 90 est une abstraction de niveau supérieur en ce sens qu'il interdit la manipulation directe d'adresse. À chaque pointeur Fortran 90 est associé un « descripteur interne » contenant les caractéristiques (type, rang, état, adresse de la cible, et même le pas d'adressage en cas de section régulière, etc . . .). Pour toute référence à ce pointeur, l'indirection est faite pour vous, d'où la notion d'« alias ». Comme nous allons le voir, ce niveau d'abstraction supérieur ne limite en rien (bien au contraire) les applications possibles.*

Déclaration d'un pointeur

Attribut `pointer` spécifié lors de sa déclaration.

Exemple

```
real, pointer                                  :: p1
integer, dimension(:), pointer                 :: p2
character(len=80), dimension(:,:), pointer :: p3
character(len=80), pointer :: p4(:)
-----------------------------------------
type boite
   integer i
   character(len=5) t
end type
type(boite), pointer  :: p5
-----------------------------------------
```

Attention : p4 n'est pas un « tableaux de pointeurs » ! !

Note *: p4 est en fait un pointeur susceptible d'être ultérieurement associé à un tableau de rang 1 et de type « chaînes de caractères ». Bien qu'autorisée, la déclaration ci-dessus est ambiguë ; on lui préférera donc systématiquement la forme classique :*

```
character(len=80), dimension(:), pointer :: p4
```

Symbole « => »

Cible : c'est un objet pointé. Cet objet devra avoir l'attribut `target` lors de sa déclaration :

```
integer, target  :: i
```

Le symbole « => » sert à *valoriser* un pointeur. Il est binaire : op1 => op2

	Pointeur	Cible
op1	⊗	
op2	⊗	⊗

Exemple

```
integer, target  :: n
integer, pointer :: ptr1, ptr2

n = 10
ptr1 => n
ptr2 => ptr1
n = 20
print *, ptr2
```

Remarque

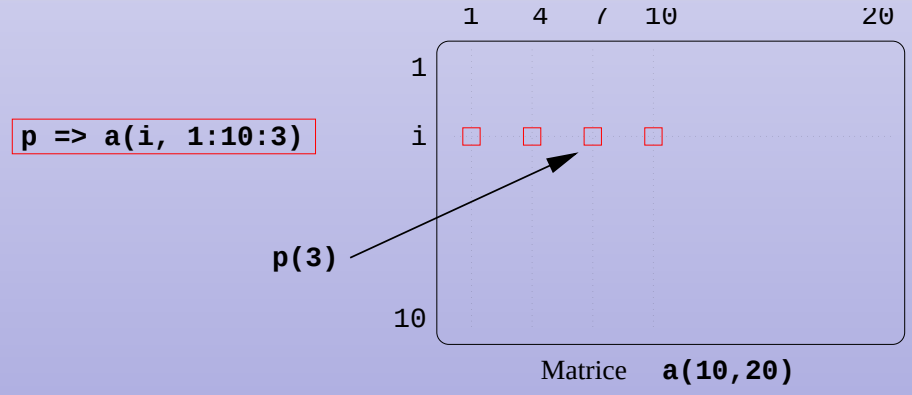
- p1 et p2 étant deux pointeurs, « p2 => p1 » implique que p2 prend l'état de p1 (indéfini, nul ou associé à la même cible).

Un pointeur peut être un alias d'objets plus complexes :

Exemple

```
implicit none
real, dimension(10,20), target :: a
real, dimension(:), pointer    :: p
integer                        :: i

!a = reshape(source=(/ (i, i=1,200) /), shape=[ 10,20 ])
! L'appel suivant est préférable :
a = reshape(source=(/ (i, i=1,200) /), shape=shape(a))
read(*, *) i
p => a(i, 1:10:3) ! p est maintenant un vecteur de profil (/ 4 /)
print *, p(3)
end
```



Symbole « = » appliqué aux pointeurs

Attention : lorsque les opérandes du symbole = sont des pointeurs, l'affectation s'effectue sur les cibles et non sur les pointeurs.

Exemple

```
implicit none
integer                          :: i
integer, pointer                 :: ptr1, ptr2
integer, target                  :: i1, i2
real, dimension(3,3), target     :: a, b
real, dimension(:,:), pointer    :: p, q
!-----------------------------------------
i1 = 1 ; i2 = 2
ptr1 => i1; ptr2 => i2
ptr2 = ptr1       ! équivalent à "i2 = i1"
print *, i2
!-----------------------------------------
a = reshape(source=(/ (i, i=1,9) /), shape=shape(a))
p => a; q => b
q = p + 1.    ! ou  q = a + 1.
print *, b
end
```

Allocation dynamique de mémoire

L'instruction ALLOCATE permet d'associer un pointeur et d'allouer dynamiquement de la mémoire.

Exemple

```
implicit none
integer, dimension(:,:), pointer :: p
integer                          :: n
read(*, *) n
ALLOCATE(p(n, n))
p = reshape(source=(/ (i, i=1,n*n) /), shape=shape(p))
print *,p
DEALLOCATE(p)
end
```

Remarques

- L'espace alloué n'a pas de nom, on y accède par l'intermédiaire du pointeur.
- Pour libérer la place allouée on utilise l'instruction DEALLOCATE
- Après l'exécution de l'instruction DEALLOCATE le pointeur passe à l'état **nul**.
- L'instruction DEALLOCATE appliquée à un pointeur dont l'état est indéterminé provoque une erreur.
- Possibilité d'allocation dynamique d'un scalaire ou d'une structure de données via un pointeur : application aux listes chaînées (cf. page 61 et 165).

Imbrication de zones dynamiques

Dans le cas de zones dynamiques imbriquées, on prendra garde à libérer ces zones convenablement. Il suffira d'appliquer la règle suivante :

Les zones seront libérées dans l'ordre inverse de celui qui a servi à les allouer.

Exemple

```
program zones_dyn
  implicit none
  type vecteur
    real x, y, z
  end type vecteur
  type champs_vecteurs
    integer n
    type(vecteur), dimension(:), pointer :: champs
  end type champs_vecteurs
  type(champs_vecteurs), dimension(:), allocatable :: tab_champs
  integer nb_elts, i
```

Exemple (suite)

```
  read *,nb_elts
  allocate(tab_champs(nb_elts))
  do i=1,nb_elts
    read *,tab_champs(i)%n
    allocate(tab_champs(i)%champs(tab_champs(i)%n))
        ...
  end do
    ...
  do i=1,nb_elts
    deallocate(tab_champs(i)%champs)
  end do
  deallocate(tab_champs)
    ...
end program zones_dyn
```

Fonction NULL() et instruction NULLIFY

Au début d'un programme un pointeur n'est pas défini : son **état** est indéterminé.
La fonction intrinsèque NULL() (**Norme 95**) permet de forcer un pointeur à l'état nul (y compris lors de sa déclaration).

Exemple

```
real, pointer, dimension(:) :: p1 => NULL()
        ...
p1 => NULL()
        ...
```

L'instruction NULLIFY permet de forcer un pointeur à l'état **nul** :

```
real, pointer :: p1, p2

nullify(p1)
nullify(p2)
        ...
```

Remarques

- Si deux pointeurs p1 et p2 sont alias de la même cible, NULLIFY(p1) force le pointeur p1 à l'état **nul**, par contre le pointeur p2 reste alias de sa cible.
- Si p1 est à l'état **nul**, l'instruction « p2 => p1 » force p2 à l'état **nul**.

Fonction intrinsèque ASSOCIATED

Il n'est pas possible de comparer des pointeurs, c'est la fonction intrinsèque ASSOCIATED qui remplit ce rôle :

```
ASSOCIATED(pointer[, target])
```

ASSOCIATED(p)	→	vrai si p est associé à une cible
	→	faux si p est à l'état **nul**
ASSOCIATED(p1, p2)	→	vrai si p1 et p2 sont alias de la même cible
	→	faux sinon
ASSOCIATED(p1, c)	→	vrai si p1 est alias de la cible c faux sinon

Remarques

- l'argument optionnel TARGET peut être au choix une cible ou un pointeur ;
- le pointeur ne doit pas être dans l'état indéterminé ;
- si p1 et p2 sont à l'état **nul** alors ASSOCIATED(p1,p2) renvoie *faux*.

Situations à éviter

Exemple 1

```
implicit none
real, dimension(:,:), pointer :: p1, p2
integer                       :: n

read(*, *) n
allocate(p2(n, n))
p1 => p2
deallocate(p2)
    ...
```

Dès lors l'utilisation de p1 peut provoquer des résultats imprévisibles.

Exemple 2

```
implicit none
real, dimension(:,:), pointer :: p
integer                       :: n

read(*, *) n
allocate(p(n, 2*n))
p = 1.
p => NULL()
    ...
```

La « zone anonyme » allouée en mémoire grâce à l'instruction ALLOCATE n'est plus référençable !

Exemple 3 : erroné

```
implicit none
real, dimension(:,:), &
      allocatable,    &
      target  :: mat
real, dimension(:,:), &
      pointer :: p => NULL()

p => mat
    ...
allocate(mat(10,20))
p = ...
    ...
```

Exemple 3 : correct

```
implicit none
real, dimension(:,:), &
      allocatable,    &
      target  :: mat
real, dimension(:,:), &
      pointer :: p => NULL()

allocate(mat(10,20))
p => mat
p = ...
    ...
```

Remarques

- un tableau ayant l'attribut ALLOCATABLE peut être la cible d'un pointeur ;
- avant d'effectuer l'association, il faut bien prendre garde à ce que le tableau soit alloué.

Déclaration de « tableaux de pointeurs »

Exemple d'utilisation d'un « **tableau de pointeurs** » pour trier (sur la sous-chaîne correspondant aux caractères 5 à 9) un tableau de chaînes de caractères de longueur 256 :

Tri avec « tableaux de pointeurs »

```
module chaine
 type ptr_chaine
   character(len=256), pointer :: p => null()
 end type ptr_chaine
end module chaine

program tri_chaine
 use chaine
 implicit none
 type(ptr_chaine), &
     dimension(:), allocatable  :: tab_pointeurs
 character(len=256), target, &
     dimension(:), allocatable  :: chaines
 integer                        :: nbre_chaines
 integer                        :: i, j

 print *,"Entrez le nombre de chaînes :"
 read(*, *) nbre_chaines
 allocate(tab_pointeurs(nbre_chaines),chaines(nbre_chaines))
```


Tri avec « tableaux de pointeurs » (suite)

```
 do i=1,nbre_chaines
   print *,"Entrez une chaîne : "; read(*,*) chaines(i)
   tab_pointeurs(i)%p => chaines(i)
 end do
 !
 do i=nbre_chaines-1,1,-1
   do j=1,i
     if (tab_pointeurs(j)%p(5:9)>tab_pointeurs(j+1)%p(5:9)) then
       !----- Permutation des deux associations -----
       tab_pointeurs(j:j+1) = tab_pointeurs(j+1:j:-1)
       !-------------------------------------------
     end if
   end do
 end do
 print "(/, a)","Liste des chaînes triées :"
 print "(a)", (tab_pointeurs(i)%p, i=1,size(tab_pointeurs))
 deallocate(chaines, tab_pointeurs)
end program tri_chaine
```

Note : l'affectation entre structures implique l'association des composantes de type pointeur, d'où l'écriture très simplifiée de la permutation sous forme d'une simple affectation.

Passage d'un pointeur en argument de procédure

1. L'argument muet n'a pas l'attribut `pointer` :
   - le pointeur doit être associé avant l'appel,
   - c'est l'adresse de la cible associée qui est passée,
   - l'interface peut être implicite ce qui permet l'appel d'une procédure Fortran 77. **Attention** : dans ce cas si la cible est une section régulière non contiguë, le compilateur transmet une copie contiguë, d'où un impact possible sur les performances (cf. chap. 5 page 89).
2. L'argument muet a l'attribut `pointer` :
   - le pointeur n'est pas nécessairement associé avant l'appel (avantage par rapport à `allocatable`),
   - c'est l'adresse du *descripteur du pointeur* qui est passée,
   - l'interface doit être explicite (pour que le compilateur sache que l'argument muet a l'attribut `pointer`),
   - si le pointeur passé est associé à un tableau avant l'appel, les bornes inférieures/supérieures de chacune de ses dimensions sont transmises à la procédure ; elles peuvent alors être récupérées via les fonctions `UBOUND/LBOUND`.

Passage d'une cible en argument de procédure

L'attribut `target` peut être spécifié soit au niveau de l'argument d'appel, soit au niveau de l'argument muet, soit au niveau des deux. Si l'argument muet a l'attribut `target`, l'interface doit être explicite :

1. si l'argument muet avec l'attribut `target` est un scalaire ou un tableau à profil implicite sans l'attribut `contiguous` et l'argument d'appel a également l'attribut `target` différent d'une section irrégulière alors :
   - tout pointeur associé à l'argument d'appel devient associé à l'argument muet ;
   - au retour de la procédure, tout pointeur associé à l'argument muet reste associé à l'argument d'appel.
2. si l'argument muet avec l'attribut `target` est un tableau à profil explicite, un tableau à profil implicite avec l'attribut `contiguous` ou un tableau à taille implicite et que l'argument d'appel a également l'attribut `target` différent d'une section irrégulière et référençant des éléments non contigus alors [1] :
   - le fait que tout pointeur associé à l'argument d'appel devienne associé à l'argument muet dépend du compilateur ;
   - de même, au retour de la procédure, l'état d'un pointeur associé dans la procédure à l'argument muet est dépendant du compilateur.
3. si l'argument muet a l'attribut `target` et l'argument d'appel n'a pas l'attribut `target` ou est une section irrégulière [1] :
   - tout pointeur associé à l'argument muet dans la procédure devient indéfini au retour de la dite procédure.

1. Attention à l'utilisation de pointeurs *globaux* ou *locaux permanents* (`save`) éventuellement associés dans la procédure à cette cible dans le cas où le compilateur aurait déclenché le mécanisme de *copy in–copy out* (cf. chapitre 5 page 89).

Passage d'une cible en argument de procédure

De plus, la norme 2008 précise que si l'argument muet a l'attribut `pointer` avec la vocation `intent(in)` et l'argument d'appel a l'attribut `target`, alors à l'entrée de la procédure l'argument muet devient associé à l'argument d'appel comme le montre l'exemple suivant :

```
module sp_module
  implicit none
contains
  subroutine sp(p)
    real, dimension(:), pointer, intent(in) :: p

    if (associated(p)) print '(*(f6.3),/)', p
  end subroutine sp
end module sp_module

program main
  use sp_module
  implicit none
  real, dimension(:), allocatable, target :: vec

  allocate(vec(10))
  call random_number(vec)
  print '(*(f6.3),/)', vec

  call sp(vec)
  deallocate(vec)
end program main
```


Passage d'une cible en argument de procédure

Autre exemple

```
module sp_module
  implicit none
  real, dimension(:,:), pointer :: p
contains
  subroutine sp(a)
    real, dimension(:,:), target :: a

    call random_number(a) ; p => a
  end subroutine sp
end module sp_module

program main
  use sp_module
  implicit none
  integer n, m
  real, dimension(:,:), allocatable, target :: a

  read(*,*) n, m ; allocate( a(n,m) )
  call sp(a(::2,:))
  do i=1,size(p,1)
    write( *, '(*(f6.2))') p(i,:)
  end do
  deallocate(a)
end program main
```

Pointeur, tableau à profil différé et COMMON

Exemple

```
real,allocatable,dimension(:,:)   :: P ! autorisé depuis 2003
real,pointer,     dimension(:,:)   :: P
real,target ,     dimension(10,10):: T1, T2, TAB
common /comm1/ P, T1,T2
    ...
P => T1   ! associé avec un tableau du common
    ...
P => TAB ! associé avec un tableau local
    ...
allocate(P(50,90)) ! P : alias zone anonyme (50x90)
```

- L'attribut ALLOCATABLE est interdit pour un tableau figurant dans un COMMON.
- Quelle que soit l'unité de programme où il se trouve, un pointeur appartenant à un COMMON doit forcément être de même type et de même rang. Le nom importe peu. Il peut être associé à un tableau existant ou à un tableau alloué dynamiquement. Cette association est connue par toute unité de programme possédant ce COMMON ;
- Attention : après chacune des deux dernières associations ci-dessus, seul le pointeur P fait partie du COMMON (pas la cible).

Liste chaînée

Exemple : liste chaînée

```
module A
  type cel
    real, dimension(4) :: x
    character(len=10)  :: str
    type(cel), pointer :: p => null()
  end type cel
  type(cel), pointer   :: debut => null()
contains
  recursive subroutine listage(ptr)
    type(cel), pointer :: ptr
    if(associated(ptr%p)) call listage(ptr%p)
    print *, ptr%x, ptr%str
  end subroutine listage
  recursive subroutine libere(ptr)
    type(cel), pointer :: ptr
    if(associated(ptr%p)) call libere(ptr%p)
    deallocate(ptr)
  end subroutine libere
end module A
```

Exemple : liste chaînée (suite)

```
program liste
  use A
  implicit none
  type(cel), pointer :: ptr_courant, ptr_precedent
  do
   if (.not.associated(debut)) then
    allocate(debut) ; ptr_courant => debut
   else
    allocate(ptr_courant); ptr_precedent%p => ptr_courant
   end if
   read  *, ptr_courant%x, ptr_courant%str
   ptr_precedent => ptr_courant
   if (ptr_courant%str == "fin") exit
  end do
  call listage(debut) ! => Impression de la dernière à la 1ère.
  call libere(debut)  ! => Libération totale de la liste.
end program liste
```

Interface implicite : définition

L'**interface de procédure** est constituée des informations permettant la communication entre deux procédures. Principalement :

- arguments d'appel (*actual arguments*),
- arguments muets (*dummy arguments*),
- instruction `function` ou `subroutine`.

En fortran 77
Compte tenu du principe de la compilation séparée des procédures et du passage des arguments par adresse, l'interface contient peu d'informations d'où une *visibilité* très réduite entre les deux procédures et donc des possibilités de contrôle de cohérence très limitées. On parle alors d'**interface « implicite »**.

En Fortran 90
Interface **« implicite »** par défaut entre deux procédures externes avec les mêmes problèmes ⇒ cf. exemple ci-après montrant quelques erreurs classiques non détectées à la compilation.

L'exemple suivant fait appel à un sous-progr. externe `maxmin` pour calculer les valeurs max. et min. d'un vecteur `vect` de taille `n` et optionnellement le rang `rg_max` de la valeur max. avec mise-à-jour de la variable de contrôle `ctl`.

Exemple

```
program inout
  implicit none
  integer, parameter :: n=100
  real,dimension(n)  :: v
  real               :: xv,y
           ...
  call maxmin(v,n,vmax,vmin,control,rgmax) ! OK
  !--->> Argument constante numérique : DANGER...
  call maxmin(v,n,vmax,vmin,0,rgmax)
  nul=0; print *, ' nul=',nul
  !--->> Erreur de type et scalaire/tableau
  call maxmin(xv,n,vmax,vmin,control,rgmax)
  !--->> Interversion de deux arguments
  call maxmin(v,n,vmax,vmin,rgmax,control)
  !--->> "Oubli" de l'argument rg_max
  call maxmin(v,n,vmax,vmin,control)
  !--->> Argument y en trop
  call maxmin(v,n,vmax,vmin,control,rgmax,y)
end program inout
subroutine maxmin(vect,n,v_max,v_min,ctl,rg_max)
  real,dimension(n)  :: vect
  V=v_max+...  !-Erreur: v_max en sortie seulement.
  n=....       !-Erreur: n en entrée seulement
  ctl=99       !-Erreur: constante passée en argument.
           ...
```


Arguments : attributs INTENT et OPTIONAL

Un meilleur contrôle par le compilateur de la cohérence des arguments est possible en Fortran 90 à deux conditions :

1. améliorer la *visibilité* de la fonction appelée. Par exemple, en la définissant comme interne (CONTAINS). On parle alors d'**interface « explicite »**.
2. préciser la vocation des arguments muets de façon à pouvoir contrôler plus finement l'usage qui en est fait. Pour ce faire, Fortran 90 a prévu :
   - l'attribut INTENT d'un argument :
     - INTENT(in) : entrée seulement ;
     - INTENT(out) : sortie seulement (dans la procédure, l'argument muet doit être défini avant toute référence à cet argument) ;
     - INTENT(inout) : entrée et sortie.

```
real, dimension(:), intent(in) :: vect
```

   - l'attribut OPTIONAL pour déclarer certains arguments comme **optionnels** et pouvoir tester leur présence éventuelle dans la liste des arguments d'appel (fonction intrinsèque PRESENT).

```
integer, optional, intent(out) :: rg_max
.....
if ( present(rg_max) ) then ...
```

Remarques

- lors de la déclaration des arguments muets d'une procédure, la vocation (attribut `INTENT`) est interdite au niveau :
  - de la valeur retournée par une fonction,
  - d'un argument de type procédural.
- `INTENT`(inout) n'est pas équivalent à l'absence de vocation ; par exemple, une constante littérale ne peut jamais être associée à un argument muet ayant l'attribut `INTENT`(inout) alors qu'elle peut l'être à l'argument sans vocation si ce dernier n'est pas redéfini dans la procédure ;
- un argument muet protégé par la vocation `INTENT`(in) doit conserver cette protection dans les autres procédures auxquelles il est susceptible d'être transmis ;
- depuis la norme 2003, il est permis de préciser la vocation aux arguments muets ayant l'attribut `POINTER` ; c'est alors l'association qui est concernée et non la cible :
  - `INTENT(IN)` : le pointeur ne pourra ni être associé, ni mis à l'état nul, ni alloué au sein de la procédure ;
  - `INTENT(OUT)` : le pointeur est forcé à l'état indéfini à l'entrée de la procédure ;
  - `INTENT(INOUT)` : le pointeur peut à la fois transmettre une association préétablie et retourner une nouvelle association.

Passage d'arguments par mot-clé

À l'appel d'une procédure, il est possible de passer des arguments par **mots-clé** ou de panacher avec des arguments positionnels. Pour la prise en compte des arguments optionnels, il est recommandé d'utiliser le passage par mots-clé. Le panachage reste possible sous deux conditions :

1. les arguments positionnels doivent toujours précéder ceux à mots-clé,
2. parmi les arguments positionnels, seuls les derniers pourront alors être omis s'ils sont optionnels.

Exemples supposant `rg_max` avec l'attribut `OPTIONAL` :

```
call maxmin(vect=v, v_max=vmax, v_min=vmin, ctl=control, rg_max=rgmax)
call maxmin(v, vmax, vmin, ctl=control)
call maxmin(v, vmax, ctl=control, v_min=vmin)
```

L'exemple suivant fait appel au sous-programme `maxmin` avec **interface « explicite »** du fait de son utilisation comme `procédure interne`. Les erreurs de cohérence signalées plus haut seraient toutes détectées à la compilation.

Procédure interne

Exemple

```
program inout
  implicit none
  integer, parameter  :: n=5
  integer             :: rgmax=0,control=0
  real, dimension(n)  :: v=(/ 1.,2.,9.,4.,-5. /)
  real                :: vmax,vmin
  call maxmin( v, vmax, vmin, control, rgmax )
  !---- Appel sans l'argument optionnel rgmax
  call maxmin( v, vmax, vmin, control )
  !---- Idem avec panachage
  call maxmin( v, vmax, ctl=control, v_min=vmin )
contains
  subroutine maxmin( vect,v_max,v_min,ctl,rg_max )
    real, dimension(:), intent(in)    :: vect
    real,               intent(out)   :: v_max, v_min
    integer, optional,  intent(out)   :: rg_max
    integer,            intent(inout) :: ctl
    v_max=MAXVAL(vect); v_min=MINVAL(vect); ctl=1
    if( present(rg_max) ) then
      rg_max=MAXLOC(vect, DIM=1); ctl=2
    end if
  end subroutine maxmin
end program inout
```


Expliciter l'interface via une **procédure interne** est une solution simple et permet bien de résoudre à la compilation tous les cas d'erreurs signalés. Elle présente néanmoins des inconvénients qui en limitent l'utilisation :

- la procédure interne n'est pas *visible* de l'extérieur,
- programmation lourde et non modulaire.

Nous verrons plus loin qu'il existe cinq solutions pour profiter de la fiabilité associée à l'interface explicite.

Remarques

- il n'est pas possible d'imbriquer les procédures internes ; l'instruction `CONTAINS` ne peut apparaître qu'une seule fois ;
- les procédures internes et la procédure les contenant forment une même et unique *scoping unit* ;
- l'instruction `IMPLICIT NONE` précisée pour une procédure s'applique également à ses procédures internes ;
- si l'instruction `IMPLICIT NONE` est spécifiée dans la partie « **data** » (*specification part*) d'un module, celle-ci n'est pas exportable via l'instruction `USE` ;
- dans une procédure interne, une variable déjà présente dans l'appelant est :
  - **globale** si elle n'est pas explicitement redéclarée,
  - **locale** si elle est explicitement redéclarée.

  D'où l'intérêt particulier de préciser l'instruction `IMPLICIT NONE` pour des procédures avant leur conversion en procédures internes Fortran 90. La nécessité de tout déclarer évite alors le risque de « **globaliser** » à tort des variables locales homonymes.

Interface explicite : 5 possibilités

1. procédures intrinsèques,
2. procédures internes (`CONTAINS`),
3. présence du **bloc interface** dans la procédure appelante,
4. la procédure appelante accède (`USE`) au module contenant le **bloc interface** de la procédure appelée,
5. la procédure appelante accède (`USE`) au module contenant la procédure appelée.

Le cas 2 a déjà été traité et commenté dans l'exemple précédent ; les cas 3, 4 et 5 seront exploités ci-après en adaptant ce même exemple.

Interface explicite avec bloc interface

Pour éviter les inconvénients de la procédure interne tout en conservant la fiabilité de l'interface « *explicite* », Fortran 90 offre la solution du `bloc interface` qui permet de donner là où il est présent une *visibilité* complète sur l'interface d'une procédure externe. Ce bloc interface peut être créé par copie de la partie déclarative des arguments muets de la procédure à interfacer. Il sera inséré dans chaque unité de programme faisant référence à la procédure externe.

Avec cette solution la procédure reste bien externe (modularité), mais il subsiste la nécessité de dupliquer le bloc interface (dans chaque procédure appelante) avec les risques que cela comporte... Par ailleurs le contrôle de cohérence est fait entre les arguments d'appel et les arguments muets définis dans le bloc interface et non pas ceux de la procédure elle-même !

Exemple avec bloc interface

```
program inout
  implicit none
  integer,parameter   :: n=5
  integer             :: rgmax=0,control=0
  real,dimension(n)   :: v=(/ 1.,2.,40.,3.,4. /)
  real                :: vmax,vmin
  !-------- Bloc interface-------------------------
  interface
   subroutine maxmin( vect, v_max, v_min, ctl, rg_max )
    real,dimension(:), intent(in)    :: vect
    real,              intent(out)   :: v_max,v_min
    integer, optional, intent(out)   :: rg_max
    integer,           intent(inout) :: ctl
   end subroutine maxmin
  end interface
  !-----------------------------------------------
  .......
  call maxmin( v, vmax, vmin, control, rgmax )
  .......
end program inout
```

Exemple avec bloc interface (suite)

```
subroutine maxmin( vect, v_max, v_min, ctl, rg_max )
  implicit none
  real,dimension(:), intent(in)    :: vect
  real,              intent(out)   :: v_max,v_min
  integer, optional, intent(out)   :: rg_max
  integer,           intent(inout) :: ctl
  v_max = MAXVAL(vect)
  v_min = MINVAL(vect)
  ctl = 1
  if( present(rg_max) ) then
    rg_max = MAXLOC( vect, DIM=1 )
    ctl = 2
  endif
  print *,"Taille vecteur via size :",SIZE(vect)
  print *,"Profil vecteur via shape:",SHAPE(vect)
end subroutine maxmin
```

Interface explicite : ses apports

- la transmission du **profil** et de la **taille** des tableaux à profil implicite et la possibilité de les récupérer via les fonctions `SHAPE` et `SIZE`,
- la possibilité de **contrôler** la vocation des arguments en fonction des attributs `INTENT` et `OPTIONAL` : en particulier l'interdiction de passer en argument d'appel une constante (type `PARAMETER` ou numérique) si l'argument muet correspondant a la vocation `OUT` ou `INOUT`,
- la possibilité de tester l'absence des arguments optionnels (fonction `PRESENT`),
- le passage d'arguments par mot-clé,
- la détection des erreurs liées à la non cohérence des arguments d'appel et des arguments muets (type, attributs et nombre) ; conséquence fréquente d'une faute de frappe, de l'oubli d'un argument non optionnel ou de l'interversion de deux arguments.

Interface explicite : module et bloc interface

Pour améliorer la fiabilité générale du programme et s'assurer d'une parfaite homogénéité du contrôle des arguments il faut insérer le même bloc interface dans toutes les unités de programme faisant référence à la procédure concernée (le sous-programme maxmin dans notre exemple).

C'est là le rôle du **module** et de l'instruction `USE` permettant l'accès à son contenu dans une unité de programme quelconque.

Un **module** est une unité de programme particulière introduite en Fortran 90 pour *encapsuler* entre autres :

- des données et des définitions de types dérivés,
- des blocs interfaces,
- des procédures (après l'instruction `CONTAINS`),

Quel que soit le nombre d'accès (`USE`) au même module, les entités ainsi définies sont uniques (remplace avantageusement la notion de `COMMON`).

Doit être compilé séparément avant de pouvoir être utilisé.

Afin de réaliser une interface « explicite », les exemples suivant font l'utilisation d'un module renfermant le bloc interface dans le 1er puis la procédure elle-même (solution la plus sûre).

Exemple avec module et bloc interface

```
module bi_maxmin
  interface
   subroutine maxmin(vect,v_max,v_min,ctl,rg_max)
     real,dimension(:), intent(in)    :: vect
     real,              intent(out)   :: v_max,v_min
     integer, optional, intent(out)   :: rg_max
     integer,           intent(inout):: ctl
   end subroutine maxmin
  end interface
end module bi_maxmin
```

Ce module est compilé séparément et stocké dans une bibliothèque personnelle de modules. Son utilisation ultérieure se fera comme dans l'exemple ci-dessous :

```
program inout
  USE bi_maxmin !<<<- Accès au bloc interface---
  implicit none
  integer,parameter   :: n=5
  integer             :: rgmax=0,control=0
  real,dimension(n)   :: v=(/ 1.,2.,40.,3.,4. /)
  real                :: vmax,vmin

  call maxmin(v, vmax, vmin, control, rgmax)
end program inout
```


Exemple avec module procédure

```
module mpr_maxmin
contains
 subroutine maxmin(vect,v_max,v_min,ctl,rg_max)
  implicit none
  real,dimension(:), intent(in)    :: vect
  real,              intent(out)   :: v_max,v_min
  integer, optional, intent(out)   :: rg_max
  integer,           intent(inout) :: ctl
  v_max=MAXVAL(vect) ; v_min=MINVAL(vect)
  ctl=1
  if(present(rg_max))then
    rg_max=MAXLOC(vect, DIM=1); ctl=2
  endif
 end subroutine maxmin
end module mpr_maxmin
```

Après compilation séparée du module, on l'utilisera comme suit :

```
program inout
  USE mpr_maxmin !<<<- Accès au module-procedure
  .......
  call maxmin(v, vmax, vmin, control, rgmax)
```

Note : l'interface est automatiquement explicite entre les procédures présentes au sein d'un même module.

Cas d'interface explicite obligatoire

Il est des cas où l'interface d'appel doit être « explicite » :

- fonction à valeur tableau,
- fonction à valeur pointeur,
- fonction à valeur chaîne de caractères dont la longueur est déterminée dynamiquement,
- tableau à profil implicite,
- argument muet avec l'attribut `allocatable`, `pointer` ou `target`,
- passage d'arguments à **mots-clé**,
- argument optionnel,
- procédure générique,
- surcharge ou définition d'un opérateur,
- surcharge du symbole d'**affectation**.

Exemple de fonctions à valeur tableau/pointeur/chaîne

```
module M1
  implicit none
contains
  function f_t(tab)      !<=== à valeur tableau
    real, dimension(:), intent(in) :: tab
    real, dimension(size(tab) + 2) :: f_t
    f_t(2:size(tab)+1) = sin(abs(tab) - 0.5)
    f_t(1) = 0. ; f_t(size(tab) + 2) = 999.
  end function f_t
  function f_p(tab, lx) !<=== à valeur pointeur
    real, dimension(:), intent(in) :: tab
    integer,            intent(in) :: lx
    real, dimension(:), pointer    :: f_p
    allocate(f_p(lx)) ; f_p = tab(1:lx*3:3) + tab(2:lx*5:5)
  end function f_p
  function f_c(str)      !<=== à valeur chaîne
    character(len=*), intent(in) :: str
    character(len=len(str))      :: f_c
    integer                      :: i
    do i=1,len(str)
      f_c(i:i)=achar(iachar(str(i:i)) - 32)
    end do
  end function f_c
end module M1
```

Exemple de fonctions à valeur tableau/pointeur/chaîne (suite)

```
program ex2
  use M1
  implicit none
  integer                          :: i
  integer, parameter               :: n = 100
  real, dimension(n)               :: t_in
  real, dimension(size(t_in)+2)    :: t_out
  real, dimension(:), pointer      :: ptr

  call random_number(t_in)
  !------ Appel fonction retournant un tableau
  t_out  = f_t(tab=t_in)
  print *, t_out( (/1, 2, 3, 99, 100, 101 /) )
  !------ Appel fonction retournant un pointeur
  ptr => f_p(tab=t_in, lx=10)
  print *, ptr; deallocate(ptr)
  !------ Appel fonction retournant une chaîne
  print *, f_c(str="abcdef")
end program ex2
```

Remarque la norme Fortran interdit la re-spécification de l'un quelconque des attributs (hormis PRIVATE ou PUBLIC) d'une entité *vue* par « USE *association* ». Le **type**, partie intégrante des attributs, est concerné :

Exemple

```
module A
 contains
   function f(x)
     implicit none
     real, intent(in) :: x
     real             :: f
     f=-sin(x)
   end function f
end module A

program pg1
 USE A          !<---------- "USE association"
 implicit none! ********
!real f   <======INTERDIT : attribut "real" déjà spécifié
 real x,y      ! ********   au niveau de f dans le module A
    ...
 y=f(x)
    ...
end program pg1
```


contradictoires !

Argument de type procédural et bloc interface

Exemple

```
module fct
  implicit none              !--------------------!
contains                     ! f=argument muet de !
  function myfonc(tab, f)    ! type procédural    !
    real :: myfonc           !--------------------!
    real, intent(in), dimension(:) :: tab
    interface !<============================!
      real function f(a)          !   BLOC     !
        real, intent(in) :: a     ! INTERFACE !
      end function f              !  de "f"    !
    end interface !<=======================!
    myfonc = f( a=sum(array=tab) )
  end function myfonc
  real function f1(a)
    real, intent(in) :: a
    f1 = a + 10000.
  end function f1
  . . . Autres fonctions f2, f3, . . .
end module fct
```


Exemple (suite)

```
program P
  use fct
  implicit none
  real, dimension(10) :: t; real x
    . . .
  x = myfonc( tab=t, f=f1) ! avec arg. d'appel f1
  x = myfonc( tab=t, f=f2) ! avec arg. d'appel f2
    . . .
```

C'est la seule solution pour fiabiliser l'appel de `f` dans `myfonc`. Ne pas déclarer `f1` et `f2` comme `EXTERNAL` dans le programme `P` et ne pas exporter le bloc interface par *use association*.

Interface générique : introduction

Possibilité de regrouper une *famille* de procédures sous un nom générique défini via un `bloc interface nommé`. À l'appel de la fonction générique, le choix de la procédure à exécuter est fait automatiquement par le compilateur en fonction du nombre et du type des arguments.

Cette notion existe en Fortran 77, mais reste limitée aux fonctions intrinsèques : selon le type de `x`, pour évaluer `abs(x)`, le compilateur choisit (notion de fonction générique) :

- `iabs(x)` si `x` entier,
- `abs(x)` si `x` réel simple précision,
- `dabs(x)` si `x` réel double précision,
- `cabs(x)` si `x` complexe simple précision.

Définition d'une fonction générique `maxmin` s'appliquant aux vecteurs qu'ils soient de type réel ou de type entier ⇒ deux sous-programmes très voisins :

- `rmaxmin` si `vect` réel,
- `imaxmin` si `vect` entier,

Nous allons successivement :

1. créer les deux sous-programmes `rmaxmin` et `imaxmin`,
2. les stocker dans un module `big_maxmin`,
3. stocker dans ce même module un bloc interface *familial* de nom `maxmin` référençant les 2 sous-programmes via l'instruction : `MODULE PROCEDURE rmaxmin, imaxmin`
4. compiler ce module pour obtenir son descripteur (`big_maxmin.mod`) et son *module objet*,
5. créer un exemple d'utilisation en prenant soin de donner accès (via `USE`) au module contenant l'interface générique en tête de toute unité de programme appelant le sous-programme `maxmin`.

Exemple avec module procédure

```
module big_maxmin
 interface maxmin
   module procedure rmaxmin,imaxmin !<<<<<<
 end interface maxmin !<-- depuis la norme Fortran 95.
contains
  subroutine rmaxmin(vect,v_max,v_min,ctl,rg_max)
   implicit none
   real, dimension(:), intent(in)    :: vect
   real,               intent(out)   :: v_max,v_min
   integer,  optional, intent(out)   :: rg_max
   integer,            intent(inout) :: ctl
   v_max=MAXVAL(vect); v_min=MINVAL(vect); ctl=1
   if(present(rg_max)) then !-- fonction logique
     rg_max=MAXLOC(vect, DIM=1); ctl=2
   endif
  end subroutine rmaxmin
  subroutine imaxmin(vect,v_max,v_min,ctl,rg_max)
   implicit none
   integer, dimension(:), intent(in)    :: vect
   integer,               intent(out)   :: v_max,v_min
   integer,      optional, intent(out)   :: rg_max
   integer,               intent(inout) :: ctl
   v_max=MAXVAL(vect); v_min=MINVAL(vect); ctl=1
   if(present(rg_max)) then !-- fonction logique
     rg_max=MAXLOC(vect, DIM=1); ctl=2
   endif
  end subroutine imaxmin
end module big_maxmin
```


Exemple avec module procédure (suite)

```
program p
  USE big_maxmin  !<<<--Accès au bloc interface et aux procédures
  implicit none
  integer, parameter :: n=5
  real,dimension(n)  :: v=(/ 1.,2.,40.,3.,4. /)
  real               :: vmax, vmin
  integer            :: control, rgmax

  call maxmin(vect=v,      v_max=vmax, v_min=vmin, &
              ctl=control, rg_max=rgmax)
          . . .
  call sp(n+2)
end program p
!
subroutine sp(k)
  USE big_maxmin  !<<<--Accès au bloc interface et aux procédures
  implicit none
  integer                :: k
  integer, dimension(k)  :: v_auto
  integer                :: vmax, vmin, control
          . . .
  call maxmin(vect=v_auto, v_max=vmax, v_min=vmin, ctl=control)
          . . .
end subroutine sp
```

Remarque s'il n'était pas possible de stocker tout ou partie des sous-programmes **rmaxmin**, **imaxmin**, etc. dans le module **big_maxmin**, on pourrait néanmoins les faire participer à la généricité en insérant leurs *parties déclaratives* dans le bloc interface *familial*. Par exemple :

Exemple

```
interface maxmin
 MODULE PROCEDURE imaxmin
 subroutine rmaxmin(vect,v_max,v_min,ctl,rg_max)
   real, dimension(:), intent(in)    :: vect
   real,               intent(out)   :: v_max,v_min
   integer,  optional, intent(out)   :: rg_max
   integer,            intent(inout) :: ctl
 end subroutine rmaxmin
end interface maxmin !<-- depuis la norme Fortran 95.
```

Exemple : contrôle de procédure Fortran 77

Nous allons maintenant montrer une application très particulière de l'interface générique permettant de fiabiliser l'appel d'une procédure Fortran 77 dont on ne pourrait (pour une raison quelconque) modifier ou accéder au source. L'objectif est de pouvoir l'appeler en passant les arguments d'appel par mot clé en imposant une valeur par défaut à ceux qui sont supposés optionnels et manquants.

Contrôle de procédure F77 via interface « générique et explicite »

Figure 2 – *appel classique d'un sous-progr. SP disponible sous forme d'un module objet sans contrôle inter-procédural*

Figure 3 – *idem en contrôlant le passage d'arguments via un « **bloc interface** »*

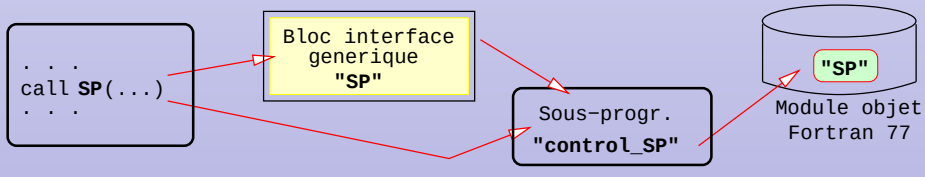


Figure 4 – *idem en utilisant un **bloc interface générique SP** appelant un sous-programme `control_SP` contrôlant l'appel de SP avec la notion d'arguments optionnels et de valeurs par défaut associées (cf. exemple ci-après).*

Exemple

```
  SUBROUTINE SP(X,I,J)
  J=I+X
  WRITE(6,*) "*** SP (F77) *** : X, I, J=", X, I, J
  END
!-----------------------------------------------
module mod1
contains
  subroutine control_SP(arg1,arg2,arg3)
   implicit none
   real,    intent(in)              :: arg1
   integer, intent(inout),optional  :: arg2
   integer, intent(out)             :: arg3
   integer                          :: my_arg2
   if(.not. present(arg2)) then !-------------------
       my_arg2 = 1              !"arg2=1" interdit !
     else                       !-------------------
       my_arg2 = arg2
   end if                       !-------------------
   call SP(arg1, my_arg2, arg3) !Appel NON générique
  end subroutine control_SP     !-------------------
end module mod1
```

Exemple (suite)

```
module module_generic
  use mod1
  interface SP                  !-----------------
   module procedure control_SP  !Bloc interface SP
  end interface SP              !-----------------
end module module_generic
!-----------------------------------------------
program prog
  use module_generic
  implicit none
  real    :: x=88.
  integer :: j                  !-----------------
  call SP(arg1=x,arg3=j)        !<-Appel générique
  print *,"Fin de prog :",x,j   !-----------------
end program prog
```

Autre solution

```
module module_generic
  interface SP                    !-----------------
   module procedure control_SP    !Bloc interface SP
  end interface SP                !-----------------
contains
  subroutine control_SP(arg1,arg2,arg3)
    real,    intent(in)                :: arg1
    integer, intent(inout), optional   :: arg2
    integer, intent(out)               :: arg3
    integer                            :: my_arg2
    interface
      subroutine SP( x, i, j )
        real,    intent(in)  :: x
        integer, intent(in)  :: i
        integer, intent(out) :: j
      end subroutine SP
    end interface
    if(.not. present(arg2)) then!--------------------
        my_arg2 = 1              !"arg2=1" interdit !
      else                       !--------------------
        my_arg2 = arg2
    end if                       !--------------------
    call SP(arg1, my_arg2, arg3)!Appel NON générique
  end subroutine control_SP      !--------------------
end module module_generic
```


Autre solution (suite)

```
program prog
 use module_generic
 implicit none
 real    :: x=88.
 integer :: j                  !-----------------
 call SP(arg1=x,arg3=j)        !<-Appel générique
 print *,"Fin de prog :",x,j   !-----------------
end program prog
```

Surcharge d'opérateurs - Introduction

Certaines notions propres aux **langages orientés objets** ont été incluses dans la norme **Fortran 90** notamment la possibilité de surcharger les opérateurs pré-définis du langage. Surcharger ou sur-définir un opérateur c'est élargir son champ d'application en définissant de nouvelles relations entre objets.

Lors de la surcharge d'un opérateur, on doit respecter sa nature (*binaire* ou *unaire*). De plus il conserve sa priorité définie par les règles de précédence du langage.

Lorsque l'on applique un opérateur à des expressions, une valeur est retournée. On emploiera donc des procédures de type `function` pour surcharger un tel opérateur.

Par contre, le symbole d'**affectation** (=), ne retournant aucune valeur, doit être sur-défini à l'aide d'une procédure de type `subroutine`.

De plus, la norme permet la définition de nouveaux opérateurs.

Il est bon de noter que le symbole d'**affectation** (=) ainsi que certains opérateurs arithmétiques et logiques ont déjà fait l'objet d'une **sur-définition** au sein du langage.

exemple

```
implicit none
integer(kind=2),parameter :: p = selected_int_kind(2)
integer(kind=p)           :: i
real,    dimension(3,3)   :: a,b,c
logical, dimension(3,3)   :: l
type vecteur
  real(kind=8)  :: x,y,z
end type vecteur
type(vecteur)   :: u,v
!------------------------------
v = vecteur( sqrt(3.)/2.,0.25,1. )
a = reshape( (/ (i,i=1,9) /),    shape=shape(a) )
b = reshape( (/ (i**3,i=1,9) /), shape=shape(b) )
    ...
c = b
u = v
l = a == b
if (a == b)... ! Incorrect POURQUOI ?
l = a < b
c = a - b
c = a * b
    ...
```

Surcharge d'opérateurs - Interface operator

Pour **surcharger un opérateur** on utilisera un bloc `interface operator`. À la suite du mot-clé `operator` on indiquera entre parenthèses le signe de l'opérateur à surcharger.

Pour **définir** un nouvel opérateur, c'est le nom (de 1 à 31 lettres) qu'on lui aura choisi encadré du caractère « **.** » qui figurera entre parenthèses.

Voici un exemple de surcharge de l'**opérateur +** :

exemple

```
module matrix
  implicit none
  type OBJ_MAT
    integer                                 :: n,m
    ! Attribut allocatable possible depuis Fortran 2003 :
    real, dimension(:,:), allocatable :: mat
  end type OBJ_MAT
  interface operator(+)
    module procedure add
  end interface
```

exemple (suite)

```
contains
  function add(a,b)
    type(OBJ_MAT), intent(in)    :: a,b
    type(OBJ_MAT)                :: add

    add%n = a%n; add%m = a%m
    allocate(add%mat(add%n,add%m))
    add%mat = a%mat + b%mat
  end function add
end module matrix
```

exemple (suite)

```
program appel
  use matrix
  implicit none
  integer       :: i, j, n, m
  type(OBJ_MAT) :: u, v, w

  print *,'Entrer la valeur de n :'
  read(*,*)n; u%n = n; v%n = n
  print *,'Entrer la valeur de m :'
  read(*,*)m; u%m = m; v%m = m
  allocate(u%mat(n,m))
  allocate(v%mat(n,m))
  u%mat = reshape( source=(/ ((real(i+j),i=1,n),j=1,m) /), &
                   shape=shape(u%mat))
  v%mat = reshape( source=(/ ((real(i*j),i=1,n),j=1,m) /), &
                   shape=shape(v%mat))
  w = u + v ! <<<<<<<<<<<<<<
  do i=1,w%n
    print *, w%mat(i,:)
  end do
end program appel
```

Surcharge d'opérateurs - Interface assignment

Pour surcharger le symbole d'**affectation** (=), utiliser un bloc interface `interface assignment`. À la suite du mot-clé `assignment` on indiquera entre parenthèses le symbole d'**affectation**, à savoir =.
Voici un exemple de surcharge du **symbole d'affectation** et de définition d'un nouvel opérateur :

exemple

```
module matrix
  implicit none
  type OBJ_MAT
    integer                                 :: n,m
    ! Attribut allocatable possible depuis Fortran 2003 :
    real, dimension(:,:), allocatable :: mat
  end type OBJ_MAT
  interface operator(+)              ! Surcharge de
    module procedure add             ! l'opérateur +
  end interface
  interface operator(.tr.)           ! Définition
    module procedure trans           ! de l'opérateur .tr.
  end interface
  interface assignment(=)            ! Surcharge de
    module procedure taille_mat      ! l'affectation
  end interface
```

exemple

```
contains
  ! Fonction associée à l'opérateur +
  function add(a,b)
    . . .
  end function add
  ! Fonction associée à l'opérateur .tr.
  function trans(a)
    type(OBJ_MAT), intent(in) :: a
    type(OBJ_MAT)             :: trans

    trans%n = a%m;trans%m = a%n
    allocate(trans%mat(trans%n,trans%m))
    trans%mat = transpose(a%mat)
  end function trans
  ! Sous-programme associé à l'affectation (=)
  subroutine taille_mat(i,a)
    integer,       intent(out) :: i
    type(OBJ_MAT), intent(in)  :: a

    i = a%n*a%m
  end subroutine taille_mat
end module matrix
```

exemple

```
program appel
  use matrix
  implicit none
  type(OBJ_MAT) :: u, v, w, t
  integer       :: taille_u, taille_v, n, m
    ...
  read *, n, m
  allocate(u%mat(n,m), v%mat(n,m), w%mat(n,m))
    ...
  !-----------
  taille_u = u
  taille_v = v
  !-----------
  t = .tr.w
end program appel
```

Remarques

- Lors de la sur-définition d'un opérateur, le ou les arguments de la fonction associée doivent avoir l'attribut `intent(in)` ;
- Lors de la sur-définition du symbole d'**affectation**, le 1erargument (opérande de gauche) doit avoir l'attribut `intent(out)` ou `intent(inout)` et le 2^{e} (opérande de droite), l'attribut `intent(in)` ;
- Les symboles => (*pointer assignment symbol*) et % (référence à une composante de structure) ne peuvent être surchargés.

Introduction

Le concepteur d'un module a la possibilité de limiter l'accès aux ressources (variables, constantes symboliques, définitions de type, procédures) qu'il se définit à l'intérieur de celui-ci. Il pourra par exemple cacher et donc rendre non exportables (via l'instruction `use`) certaines variables et/ou procédures du module.

Ceci peut se justifier lorsque certaines ressources du module ne sont nécessaires qu'à l'intérieur de celui-ci. De ce fait, le concepteur se réserve le droit de les modifier sans que les unités utilisatrices externes ne soient impactées.

Cela permettra également d'éviter les risques de conflits avec des ressources d'autres modules.

Ces ressources non exportables sont dites **privées**. Les autres sont dites **publiques**.

Par défaut, toutes les ressources d'un module (variables, procédures) sont **publiques**.

La *privatisation* de certaines données (concept d'*encapsulation de données*) conduit le concepteur à fournir au développeur des *méthodes* (procédures publiques) facilitant la manipulation globale d'objets privés ou semi-privés. Leur documentation et leur fourniture est un aspect important de la programmation objet.

Instructions PRIVATE et PUBLIC

À l'entrée d'un module le **mode par défaut** est le mode `PUBLIC`.

Les **instructions** `PRIVATE` ou `PUBLIC` *sans argument* permettent respectivement de changer de mode ou de confirmer le mode par défaut ; ce mode s'applique alors à toutes les ressources de la partie données (*specification part*) du module.

Ce type d'instruction ne peut apparaître qu'**une seule fois** dans un module.

Exemple

```
module donnee
  integer, save                  :: i   ! privée
  real, dimension(:), pointer    :: ptr ! privée
  private
  character(len=4)               :: car ! privée
end module donnee

module mod
  public
  logical, dimension(:), allocatable :: mask ! publique
    ...
end module mod
```

Attributs PRIVATE et PUBLIC

On peut définir le mode d'une ressource d'un module au moyen de l'**attribut** `PRIVATE` ou `PUBLIC` indiqué à sa déclaration.
Bien distinguer :

- l'instruction `PRIVATE` ou `PUBLIC` *sans argument* qui permet de définir le *mode* de visibilité,
- cette même instruction à laquelle on spécifie une liste d'objets auquel cas ce sont ces objets qui reçoivent l'**attribut** indiqué.

Exemple

```
module donnee
  private
  integer, public :: i, j      ! publique
  real            :: x, y, z   ! y,z : privées
  public          :: x         ! publique
  public          :: sp
contains
  subroutine sp(a,b)           ! publique
     ...
  end subroutine sp
  logical function f(x)        ! privée
     ...
  end function f
end module donnee
```

Note : pour déclarer la variable `x` publique il serait préférable de coder : `real, public :: x`

Type dérivé « semi-privé »

Les attributs précédents peuvent également s'appliquer aux types dérivés.
Un type dérivé peut être :

- **public** ainsi que ses composantes, on parle alors de type dérivé **transparent**.
- **privé**
- **public** mais avec **toutes** ses composantes **privées**. On parle alors de type dérivé « **semi-privé** ».

L'intérêt du type dérivé « **semi-privé** » est de permettre au concepteur du module le contenant d'en modifier sa structure sans en affecter les unités utilisatrices.

Par défaut les composantes d'un type dérivé `public` sont `publiques`.

Exemple

```
MODULE mod
  private :: t4
  !------------------------------
  type t1                ! semi-privé
    private
    . . . . . .
  end type t1
  !------------------------------
  type, private :: t2 ! privé
    . . . . . .
  end type t2
  !------------------------------
  type t3                ! public
    . . . . . .
  end type t3
  !------------------------------
  type t4                ! privé
    . . . . . .
  end type t4
  !------------------------------
  . . . . . .
END MODULE mod
```


Exemple complet de création d'un module au sein duquel on définit :

- des variables globales (ici `nb_lignes` et `nb_colonnes` — alternative au `COMMON`),
- un type-dérivé `OBJ_MAT` *semi-privé*,
- certaines ressources *privées*,
- des procédures de surcharge/définition d'opérateurs,
- des *méthodes* (**poubelle**, **imp**).

Exemple

```
module matrix
  integer :: nb_lignes=0, nb_colonnes=0
  !
  type OBJ_MAT
    private
    ! Initialisation des composantes possible depuis Fortran 95 :
    integer                              :: n=0,m=0
    ! Attribut allocatable possible depuis Fortran 2003 :
    real,dimension(:,:), allocatable :: mat
  end type OBJ_MAT
  !
  private :: valorisation,add,taille_mat,trans
  !-------------------------------------------
  interface operator(+)
     module procedure add
  end interface
```

Exemple (suite)

```
  interface operator(.tr.)
    module procedure trans
  end interface
  !-----------------------------------------
  interface assignment(=)
    module procedure taille_mat, valorisation
  end interface
  !-----------------------------------------
contains
  subroutine valorisation(a,t)
    type(OBJ_MAT),       intent(inout) :: a
    real, dimension(:), intent(in)    :: t

    if ( nb_lignes <= 0 .or. nb_colonnes <= 0 ) then
      print *, "Valorisation impossible."
      print *, "Veuillez valoriser le profil &
               &(nb_lignes,nb_colonnes) &
               &avec des valeurs acceptables."
    else
      if (.not.allocated(a%mat)) then
        allocate( a%mat(nb_lignes,nb_colonnes) )
        a%n = nb_lignes; a%m = nb_colonnes
      endif
      call affectation
    end if

  contains
```


Exemple (suite)

```
    subroutine affectation
      real, dimension(size(a%mat)) :: remplissage

      remplissage = 0.
      a%mat = reshape( source = t,           &
                       shape  = shape(a%mat), &
                       pad    = remplissage )
    end subroutine affectation
  end subroutine valorisation
  !-------------------------------
  subroutine poubelle(a)
    type(OBJ_MAT), intent(inout) :: a

    if (allocated(a%mat)) then
      a%n = 0; a%m = 0; deallocate(a%mat)
    endif
  end subroutine poubelle
  !-------------------------------
  function add(a,b)
    type(OBJ_MAT), intent(in) :: a, b
    type(OBJ_MAT)             :: add

    allocate(add%mat(a%n,a%m))
    add%n = a%n; add%m = a%m; add%mat = a%mat + b%mat
  end function add
```

Exemple

```
  subroutine taille_mat(i,a)
    integer,        intent(out) :: i
    type(OBJ_MAT), intent(in)  :: a
    i = a%n*a%m
  end subroutine taille_mat
  !--------------------------------
  function trans(a)
    type(OBJ_MAT), intent(in)  :: a
    type(OBJ_MAT)              :: trans
    allocate(trans%mat(a%m, a%n))
    trans%n=a%m; trans%m=a%n; trans%mat=transpose(a%mat)
  end function trans
  !--------------------------------
  subroutine imp(a)
    type(OBJ_MAT), intent(in)  :: a
    integer(kind=2)            :: i
    if ( .not.allocated(a%mat) ) then
      print *,"Impression : objet inexistant."; stop 16
    end if
    do i=1,size(a%mat,1)
      print *,a%mat(i,:)
    enddo
  end subroutine imp
end module matrix
```


Exemple d'unité utilisatrice de ce module

```
program appel
  use matrix
  implicit none
  integer       :: i, j, taille
  type(OBJ_MAT) :: u, v, w, t

  print *,"Nb. de lignes : ";  read *, nb_lignes
  print *,"Nb. de colonnes :"; read *, nb_colonnes
  u = (/ ((real(i+j),i=1,nb_lignes),j=1,nb_colonnes) /)
  v = (/ ((real(i*j),i=1,nb_lignes),j=1,nb_colonnes) /)
            ...
  u=v ! Voir remarques ci-après.
            ...
  call imp(u) ; call imp(v)
  call poubelle(v)
  taille = w  ;  call imp(w)
  call poubelle(w)
  t = .tr. u  ;  call imp(t)
  call poubelle(u)
  call poubelle(t)
end program appel
```

Remarques

- concernant l'affectation `u = v`, bien que celle-ci ait un sens il peut être intéressant d'en avoir la totale maîtrise dans le but d'effectuer un certain nombre de contrôles : pour cela on surcharge le symbole d'affectation. Sur la page suivante, on a donc ajouté au sein du bloc « `interface assignment(=)` » un sous-programme `affect` que l'on définit ensuite ;
- les objets de type `OBJ_MAT` retournés par les fonctions telles que `add` et `trans` sont temporaires : lors de leur disparition en mémoire, gérée par le compilateur, la norme `Fortran 2003` précise que les composantes ayant l'attribut `ALLOCATABLE` (ici `mat`) sont automatiquement désallouées ;
- dans le cas où l'attribut `POINTER` était indiqué à la place de `ALLOCATABLE` pour cette composante, la gestion mémoire aurait été sous la responsabilité du programmeur : pour ce faire celui-ci dispose de procédures spécifiques appelées *final procedure*. Se reporter au cours « `Fortran Expert` » pour plus de précisions.

Solution avec redéfinition de l'affectation

```
module matrix
  interface assignment(=)
    module procedure taille_mat, valorisation, affect
  end interface
!--------------------------------
contains
  . . . . .
  subroutine affect(a,b)
    type(OBJ_MAT), intent(inout) :: a
    type(OBJ_MAT), intent(in)    :: b
    if (.not.allocated(b%mat)) &
      stop "Erreur : membre de droite de  &
           &l'affectation non initialisé"
    if (allocated(a%mat)) then
      if (any(shape(a%mat) /= shape(b%mat))) &
        stop "Erreur : affect. matrices non conformantes"
    else
      allocate(a%mat(b%n,b%m), stat=etat)
      if (etat /= 0) &
        stop "Erreur ==> allocation membre de gauche"
      ! Il est parfois préférable de laisser le compilateur gérer
      ! l'erreur pour récupérer la "traceback" éventuellement plus
      ! informative. Dans ce cas, ne pas spécifier « stat=etat ».
    end if
    a%n = b%n ; a%m = b%m ; a%mat = b%mat
  end subroutine affect
  . . . . .
```

Paramètre ONLY de l'instruction USE

De même que le concepteur d'un module peut cacher des ressources de ce module, une unité utilisatrice de celui-ci peut s'interdire l'accès à certaines d'entre elles.
Pour cela on utilise le paramètre **only** de l'instruction **use**.

Exemple

```
module m
 type t1
  ...
 end type t1
 type t2
  ...
 end type t2
 logical, dimension(9) :: l
contains
 subroutine sp(...)
   ...
 end subroutine sp
 function f(...)
   ...
 end function f
end module m

program util
 use m,only : t2,f ! Seules les ressources t2 et f sont exportées
```


Lors de l'utilisation d'un module, on peut être géné par les noms des ressources qu'il nous propose, soit parce que dans l'unité utilisatrice il existe des ressources de même nom ou bien parce que les noms proposés ne nous conviennent pas.
Dans ce cas, il est possible de *renommer* les ressources du module au moment de son utilisation via le symbole « => » que l'on spécifie au niveau de l'instruction **use**. La **norme** 2003 permet de plus le renommage des **opérateurs** non intrinsèques.

Exemple

```
use m, mon_t2=>t2, mon_f=>f
use m, only : mon_t2=>t2, mon_f=>f
use m, OPERATOR(.MY_OPER.) => OPERATOR(.OPER.) ! Fortran 2003
```

Remarque

- on notera l'analogie entre ce type de *renommage* et l'affectation des *pointeurs*.

Accès aux fichiers pré-connectés

Le compilateur **Fortran** 2003 est livré avec plusieurs modules intrinsèques tel que ISO_FORTRAN_ENV qui contient des constantes symboliques permettant notamment le référencement des fichiers pré-connectés :

- INPUT_UNIT, OUTPUT_UNIT et ERROR_UNIT sont des entiers correspondant aux numéros des unités logiques relatifs à **l'entrée standard**, **la sortie standard** et à **la sortie d'erreur**. Ils remplacent avantageusement l'**astérisque** employé traditionnellement au niveau du paramètre UNIT des instructions READ/WRITE ;
- IOSTAT_END et IOSTAT_EOR sont des entiers correspondant aux valeurs négatives prises par le paramètre IOSTAT des instructions d'entrée/sortie en cas de fin de fichier ou de fin d'enregistrement. Cela permet d'enrichir la portabilité d'un code **Fortran**. Cependant, les cas d'erreurs génèrent une valeur positive restant dépendante du constructeur.

Instruction OPEN

1. NEWUNIT=
   - ce nouveau mot-clé, introduit par la **norme** 2008, permet de spécifier une variable entière qui sera valorisée à l'issue de l'OPEN à un numéro d'unité logique n'interférant pas avec ceux déjà employés, notamment ceux concernant les fichiers pré-connectés ;
2. STATUS=
   - 'REPLACE' : si le fichier n'existe pas, il sera créé, sinon il sera détruit et un fichier de même nom sera créé.
3. POSITION=
   - 'REWIND' : indique que le pointeur du fichier sera positionné à son début ;
   - 'APPEND' : indique que le pointeur du fichier sera positionné à sa fin ;
   - 'ASIS' : permet de conserver la position du pointeur du fichier ; Ne fonctionne que si le fichier est déjà connecté. C'est utile lorsque l'on désire (via l'instruction OPEN) modifier certaines caractéristiques du fichier tout en restant positionné (valeur par défaut). Très limitatif et dépendant du constructeur !
4. PAD=
   - 'YES' : des enregistrements lus avec format sont complétés avec des blancs (*padding*) dans le cas où la liste de variables à traîter et le format correspondant nécessitent plus de caractères que l'enregistrement n'en contient. (valeur par défaut).
   - 'NO' : pas de *padding*.
5. ACTION=
   - 'READ' : toute tentative d'écriture est interdite ;
   - 'WRITE' : toute tentative de lecture est interdite ;
   - 'READWRITE' : les opérations de lecture et écriture sont autorisées (valeur par défaut).
6. DELIM= $\Rightarrow$ ce paramètre permet de délimiter les chaînes de caractères **écrites** par des *namelist* ou en format libre :
   - 'APOSTROPHE' : indique que l'apostrophe « ' » sera utilisée ;
   - 'QUOTE' : indique que le guillemet « " » sera utilisée ;
   - 'NONE' : indique qu'aucun délimiteur ne sera utilisé (valeur par défaut).

Instruction INQUIRE

- RECL=*n* : permet de récupérer la longueur maximale des enregistrements.
- POSITION=*chaîne* : permet de récupérer la valeur du même paramètre spécifié lors de l'open.
- ACTION=*chaîne* : permet de récupérer la valeur du même paramètre spécifié lors de l'open.
- DELIM=*chaîne* : permet de récupérer la valeur du même paramètre spécifié lors de l'open.
- IOLENGTH=*long* : permet de récupérer la longueur de la liste des entités spécifiées. C'est utile lorsque l'on veut valoriser le paramètre RECL de l'ordre OPEN pour un fichier binaire à accès direct.
- PAD=*chaîne* : permet de récupérer la valeur du même paramètre spécifié lors de l'OPEN.

Exemples

```
inquire(9, opened=op, action=ac)
inquire(file="donnee", position=pos)
inquire(iolength=long)x,y,tab(:n)
open(2,status="scratch",action="write",access="direct",recl=long)
```

Note : l'argument IOLENGTH de l'instruction INQUIRE permet de connaître la longueur (E./S. binaires) d'une structure de type dérivé (sans composante pointeur) faisant partie de la liste spécifiée.

Entrées-sorties sur les fichiers texte (advance='no')

Le paramètre ADVANCE='no' des instructions READ/WRITE (ADVANCE='yes' par défaut) permet de rester positionner sur l'enregistrement courant.

Dans le cas d'une lecture avec format explicite et en présence du paramètre ADVANCE='no' :

- le paramètre EOR=*nnn* effectue un transfert à l'étiquette *nnn* lorsqu'une fin d'enregistrement est détectée ;
- le paramètre SIZE=*long* (*long* variable de type INTEGER) de l'instruction READ permet de récupérer le nombre de caractères transférés lors de la lecture. Dans le cas où la fin d'enregistrement est détectée, ce nombre ne tient pas compte du *padding* si *padding* il y a (paramètre PAD valorisé à 'yes' lors de l'OPEN).

Note : ADVANCE='no' est incompatible avec le format libre.

Une alternative aux paramètres END=*nnn* et EOR=*nnn* de l'instruction READ est l'emploi du paramètre IOSTAT. Il retourne un entier :

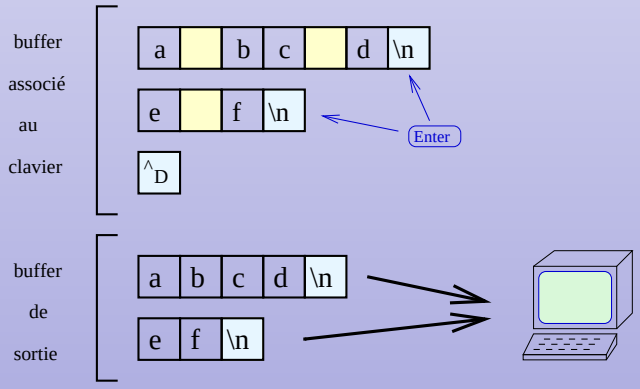
- **positif** en cas d'erreur ;
- **négatif** lorsqu'une fin de fichier ou une fin d'enregistrement est atteinte (valeurs dépendant du constructeur) ;
- **nul** sinon.

Exemple

```
read(8, fmt=9, advance="no", size=n, eor=7, end=8) list
read(8, fmt=9, advance="no", size=n, iostat=icod) list
```

Exemple

```
  character(len=1) :: c
  !-----------------------------------------------
  ! Lecture de lignes au clavier et affichage des
  ! mêmes lignes sans les blancs. Arrêt par Ctrl_D
  !-----------------------------------------------
  do
    do
      read(*,"(a)",advance="no",eor=1,end=2)   c
      if(c /= " ")write(*,"(a)",advance="no") c
    end do
1   write(*,"(a)",advance="yes")
  end do
  !-----------------------------------------------
2 print *,"Fin de la saisie."
```



Paramètres IOSTAT et IOMSG de l'instruction READ

Le module intrinsèque ISO_FORTRAN_ENV introduit par le standard 2003 définit deux fonctions **is_iostat_end**, **is_iostat_eor** permettant de gérer les évènements fin de fichier et fin d'enregistrement respectivement.
De plus, un nouveau mot-clé IOMSG précisé au niveau de l'instruction READ permet de référencer une chaîne de caractères laquelle sera valorisée en cas d'erreur au texte correspondant au traitement de l'action standard du compilateur.

Réécriture de l'exemple précédent

```
character(len=1) :: c
integer          :: ios
!-----------------------------------------------
! Lecture de lignes au clavier et affichage des
! mêmes lignes sans les blancs. Arrêt par Ctrl_D
!-----------------------------------------------
do
  read(*,"(a)",advance="no",iostat=ios)  c
  if ( is_iostat_end(ios) ) exit
  if ( is_iostat_eor(ios) ) then
    write(*,"(a)",advance="yes")
    cycle
  end if
  if(c /= " ")write(*,"(a)",advance="no") c
end do
print *,"Fin de la saisie."
```

Paramètres IOSTAT et IOMSG de l'instruction READ

Autre exemple

```
PROGRAM saisie_clavier
  implicit none
  INTEGER :: date
  INTEGER :: ios
  CHARACTER(len=256) :: errmess

  do
    print *,"Saisie d'une date :"
    read( *, '(i4)', IOSTAT=ios, IOMSG=errmess ) date
    if ( is_iostat_end(ios) ) exit
    if ( ios > 0 ) then
      print *, trim(errmess)
      print *,"Saisie invalide. Veuillez recommencer."
    else
      print '(i4)', date
    end if
  end do
  print *, "Arrêt de la saisie."
END PROGRAM saisie_clavier
```

Instruction NAMELIST

Exemple

```
integer            :: n
real,dimension(2)  :: x
character(len=5)   :: text
namelist /TRUC/ n,x,text
 ..........
read(*, nml=TRUC)
x=x+n*2
open(unit=6,delim="apostrophe")
write(6, nml=TRUC)
```

Exemples de jeux de données à lire :

```
&TRUC n=3 x=5.,0. text="abcde" /
&TRUC x=2*0.0 text="abcde" n=3 /
&TRUC text="QWERT" x=1.0 /
```

L'écriture correspondant au premier jeu de données donnerait :

```
&TRUC n=3, x=11.,6., text='abcde' /
```

Norme 95 : possibilité de commenter via le caractère « ! » des enregistrements en entrée d'une NAMELIST :

```
&TRUC x=2*0.0 ! x est un tableau
text="abcde" n=3 /
```

Note : en Fortran 77, la fin des données était en général repérée par &END ou $END au lieu de « / » et les enregistrements devaient commencer par un blanc. La relecture de données codées avec l'ancien format est soit automatiquement compatible (ex : *ifort* d'INTEL), soit assurée via :

- une variable (export XLFRTEOPTS="namelist=old" sur IBM).

Spécification de format

Norme 95 : afin de permettre l'écriture formatée de variables sans avoir à se préoccuper de la largeur du champ récepteur, il est possible de spécifier une longueur nulle avec les formats I, F, B, O et Z :

```
write(6,"(2I0,2F0.5,E15.8)") int1,int2,x1,x2,x3
```

On évite ainsi l'impression d'astérisques bien connue des programmeurs Fortran dans le cas d'un débordement de la zone réceptrice.

Procédures d'accès à l'environnement

Il est désormais possible d'accéder de façon portable aux arguments de la ligne de commande et aux variables d'environnement à l'aides de procédures.

```
GET_COMMAND( [command] [,length] [,status] )
```

- **command** est une chaîne de caractères dans laquelle sera stocké le nom de la commande (y compris les arguments s'ils existent) qui a servi à lancer l'exécutable. Elle sera valorisée avec des blancs si la récupération est impossible ;
- **length** est un entier qui sera valorisé à la longueur de la chaîne de caractères ci-dessus. Si impossibilité, 0 sera retournée ;
- **status** est un entier qui sert de code retour. Sa valeur sera :
  - 0 si l'exécution s'est bien déroulée ;
  - -1 si l'argument command a été précisée avec une taille inférieure à la longueur de la commande ;
  - > 0 si l'exécution s'est terminée en erreur.

Exemple

```
program p
  implicit none
  character(len=:), allocatable :: commande
  integer long
  call GET_COMMAND( LENGTH=long )
  allocate( character(len=long) :: commande )
  call GET_COMMAND( COMMAND=commande )
  print *, "Commande lancée ==> ", commande
  deallocate( commande )
end program p
```



```
GET_COMMAND_ARGUMENT( number [,value] [,length] [,status] )
```

- **number** est un entier qui est fourni en entrée. Il indique le numéro de l'argument désiré parmi ceux spécifiés lors du lancement de l'exécutable. La valeur 0 fait référence au nom de l'exécutable et la fonction COMMAND_ARGUMENT_COUNT retourne le nombre d'arguments qui suivent ;
- **value** est une chaîne de caractères dans laquelle sera retournée l'argument désigné ci-dessus. Une chaîne à blancs sera renvoyée si la valorisation est impossible ;
- **length** est un entier qui sera valorisé à la longueur de la chaîne de caractères ci-dessus. Si impossibilité, 0 sera retournée ;
- **status** est un entier qui sert de code retour qui est valorisé de la même façon que pour la procédure GET_COMMAND.

Exemple

```
program p
  implicit none
  character(len=:), allocatable :: arg
  integer long, i
  do i=0, COMMAND_ARGUMENT_COUNT()
    call GET_COMMAND_ARGUMENT( NUMBER=i, LENGTH=long )
    allocate( character(len=long) :: arg )
    call GET_COMMAND_ARGUMENT( NUMBER=i, VALUE=arg )
    print *,"Argument de rang ", i, " ==> ", arg
    deallocate( arg )
  end do
end program p
```

```
GET_ENVIRONMENT_VARIABLE( name [,value] [,length] [,status] [,trim_name] )
```

- **name** est une chaîne de caractères valorisée au nom de la variable d'environnement dont on désire le contenu ;
- **value** est une chaîne de caractères dans laquelle sera retournée le contenu de la variable d'environnement fournie ci-dessus. Une chaîne à blancs sera renvoyée dans les cas suivant :
  - la variable d'environnement indiquée n'existe pas ;
  - la variable d'environnement existe mais son contenu est vide ;
  - la notion de variable d'environnement n'existe pas sur la plateforme utilisée.
- **length** est un entier qui sera valorisé à la longueur de la valeur de la variable d'environnement fournie si celle-ci existe et a un contenu défini. Sinon 0 sera la valeur retournée ;
- **status** est un entier qui sert de code retour qui est valorisé de façon suivante :
  - si la variable d'environnement existe avec un contenu vide ou bien admet une valeur retournée avec succès,
    0 sera la valeur retournée ;
  - si la variable d'environnement existe et admet un contenu dont la taille est supérieure à celle de la chaîne de caractères fournie via le paramètre `value`, -1 sera retournée ;
  - ce code de retour est valorisé à 1 si la variable d'environnement n'existe pas et à 2 si il n'y a pas de notion de variable d'environnement sur la plateforme utilisée.
- **trim_name** est un logique. S'il a pour valeur `.FALSE.` alors les caractères blancs situés à la fin du paramètre **name** seront considérés comme significatifs. Ils seront ignorés dans les autres cas.

Exemple

```
program p
  implicit none
  character(len=:), allocatable :: path
  integer long

  call GET_ENVIRONMENT_VARIABLE( NAME="PATH", LENGTH=long )
  allocate( character(len=long) :: path )
  call GET_ENVIRONMENT_VARIABLE( NAME="PATH", VALUE=path )
  print *, "PATH=", path
  deallocate( path )
end program p
```

Les deux procédures suivantes permettent de récupérer des informations à propos du compilateur qui a permis la construction de l'exécutable :

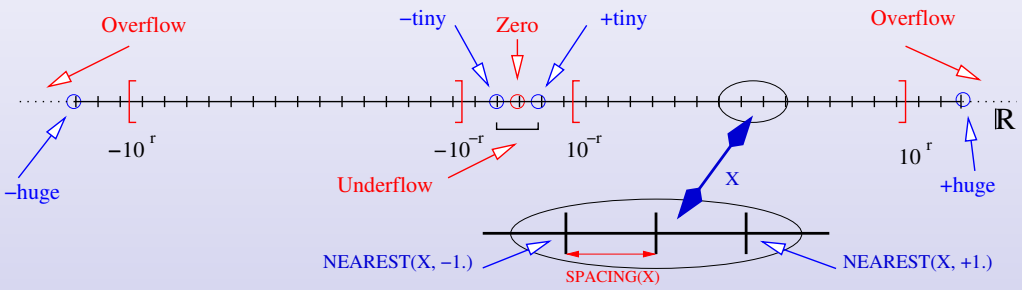
- `COMPILER_VERSION` : fonction retournant la version du compilateur utilisé ;
- `COMPILER_OPTIONS` : fonction retournant la liste des options qui ont été fournies lors de la compilation.

Exemple

```
program options
  use ISO_FORTRAN_ENV
  character(len=10) arg
  logical           mode_verbose

  mode_verbose = .false.
  do i=1,command_argument_count()
    call get_command_argument(i,arg)
    if ( arg .eq. "-v" ) then
      mode_verbose = .true.
      exit
    end if
  end do
  if ( mode_verbose ) then
    print '(a,/,a)', "Version du compilateur utilisé :", &
                     COMPILER_VERSION()
    print '(a,/,a)', "Options transmises au compilateur :", &
                     COMPILER_OPTIONS()
  end if
end program options
```


Précision et codage numérique



`TINY(x)`	plus petite valeur réelle représentable dans le sous-type de `x` (limite de l'*underflow*).
`HUGE(x)`	plus grande valeur réelle ou entière représentable dans le sous-type de `x` (limite de l'*overflow*).
`NEAREST(x, s)`	valeur réelle représentable la plus proche (à droite si `s>0.` ou à gauche si `s<0.`) de la valeur représentable correspondant à l'argument réel `x` fourni. Dépendant du sous-type de `x`.
`SPACING(x)`	écart entre deux valeurs représentables dans le sous-type de `x` au voisinage de `x`.
`EPSILON(x)`	⇒ `SPACING(+1.)` : quantité considérée comme négligeable comparée à 1.
`RANGE(x)`	c.f. chapitre 2 – Généralités (`KIND`).
`PRECISION(x)`	c.f. chapitre 2 – Généralités (`KIND`).

`SIGN(a,b)` entier/réel dont la valeur absolue est celle de `a` et le signe celui de `b`. Seule fonction distinguant +0. et -0. si ce dernier est représentable.

Note *: le zéro réel classique (+0.) a une représentation binaire totalement nulle alors que le zéro négatif (-0.) a son bit de signe positionné à 1 ('80000000' en hexa.). Seule la fonction SIGN (au niveau du 2e argument) fait la distinction entre ces deux zéros. Cette distinction peut aussi être faite via une impression en format libre. La valeur -0. est représentable sur NEC SX8 et IBM SP6.*

Mesure de temps, nombres aléatoires

`CPU_TIME(time)` (disponible depuis la `norme 95`) sous-progr. retournant dans le réel `time` le temps CPU en secondes (ou réel < 0 si indisponible). Par différence entre deux appels, il permet d'évaluer la consommation CPU d'une section de code.

`DATE_AND_TIME(date,time,zone,values)` sous-programme retournant dans les variables caractère `date` et `time`, la date et l'heure en temps d'*horloge murale*. L'écart par rapport au temps universel est retourné optionnellement dans `zone`. Toutes ces informations sont aussi stockées sous forme d'entiers dans le vecteur `values`.

`SYSTEM_CLOCK(count,count_rate,count_max)` sous-programme retournant dans des variables entières la valeur du compteur de périodes d'horloge (`count`), le nombre de périodes/sec. (`count_rate`) et la valeur maximale de ce compteur (`count_max`) ; ne permet pas d'évaluer le temps CPU consommé par une portion de programme.

`RANDOM_NUMBER(harvest)` sous-progr. retournant un/plusieurs nombres pseudo-aléatoires compris entre 0. et 1. dans un scalaire/tableau réel passé en argument (`harvest`).

`RANDOM_SEED(size,put,get)` sous-programme permettant de ré-initialiser une série de nombres aléatoires. Tous les arguments sont optionnels. En leur absence le *germe* d'initialisation dépend du constructeur. Voir exemples ci-après...

Exemple

```
real,dimension(2048,4) :: tab
call random_number(tab) ! 1ère série
 . . . .
call random_number(tab) ! 2ème série différente
 . . . .
```

Génération de deux séries identiques de nombres aléatoires dans un tableau `tab` en sauvegardant (`GET`) puis réinjectant (`PUT`) le même *germe*. La taille du vecteur `last_seed` de sauvegarde du germe est récupérée via l'argument de sortie `SIZE` :

Exemple

```
integer                              :: n
integer,allocatable,dimension(:) :: last_seed
real,dimension(2048,4)               :: tab
call random_seed(SIZE=n)
allocate(last_seed(n)) ; call random_seed(GET=last_seed)
call random_number(tab) ! 1ère série
 . . . .
call random_seed(PUT=last_seed)
call random_number(tab) ! 2ème série identique
deallocate(last_seed )
```

Attention : il est recommandé de gérer le *germe* d'initialisation uniquement via le sous-programme `RANDOM_SEED`.

Évaluation du temps CPU et du temps d'horloge (*elapsed time*)

```
INTEGER :: &
  cpt_init,& ! Val. init.  compteur périodes horloge
  cpt_fin, & ! Val. finale compteur périodes horloge
  cpt_max, & ! Valeur maximale du compteur d'horloge
  freq,    & ! Nb. de périodes d'horloge par seconde
  cpt        ! Nb. de périodes d'horloge du code
REAL :: temps_elapsed , t1, t2, t_cpu
. . .
! Initialisations
! ---------------
CALL SYSTEM_CLOCK(COUNT_RATE=freq, COUNT_MAX=cpt_max)
. . .
CALL SYSTEM_CLOCK(COUNT=cpt_init)
CALL CPU_TIME(TIME=t1)
. . .
!<<<<<<<<<<< Partie du code à évaluer >>>>>>>>>>>>
. . .
CALL CPU_TIME(TIME=t2)
CALL SYSTEM_CLOCK(COUNT=cpt_fin)
!
cpt = cpt_fin - cpt_init
IF (cpt_fin < cpt_init) cpt = cpt + cpt_max
temps_elapsed = REAL(cpt) / freq
t_cpu = t2 - t1
!
print *, "Temps elapsed = ", temps_elapsed, " sec."
print *, "Temps CPU     = ", t_cpu,          " sec."
. . .
```

Sortie de la date et de l'heure courante via la fonction intrinsèque DATE_AND_TIME

```
program date
  implicit none
  integer                  :: n
  integer, dimension(8) :: valeurs
  !
  call DATE_AND_TIME(VALUES=valeurs)
  print *
  print "(49a)", ("-", n=1,49)
  print "(a,2(i2.2,a),i4,a,3(i2.2,a),a)", &
              "| Test date_and_time ==> ", &
              valeurs(3), "/",              &
              valeurs(2), "/",              &
              valeurs(1), " - ",            &
              valeurs(5), "H",              &
              valeurs(6), "M",              &
              valeurs(7), "S", " |"
  print "(49a)", ("-", n=1,49)
end program date
```

Sortie correspondante :

```
-------------------------------------------------
| Test date_and_time ==> 14/12/2011 - 10H13M50S |
-------------------------------------------------
```

Transformation

`TRANSFER(source, mold [,size])`

⇒ scalaire ou vecteur avec représentation physique identique à celle de **source**, mais interprétée avec le type de **mold**.

- Si **size** absent, retourne un vecteur si **mold** est de rang ≥ 1 (sa taille est le plus petit nombre tel que sa représentation physique mémoire contienne celle de source), sinon un scalaire,
- Si **size** présent, retourne un vecteur de taille **size**.

Exemple

`TRANSFER( 1082130432, 1.0 )` ⇒ `4.0` (sur machine IEEE)
`TRANSFER( (/ 1.,2.,3.,4. /), (/ (0.,0.) /) )` ⇒ `(/ (1.,2.),(3.,4.) /)`

Exemple

```
integer(kind=8),dimension(4) :: tampon
character(len=8)             :: ch
real(kind=8),dimension(3)    :: y
ch    = TRANSFER( tampon(1) , "abababab" )
!y(:) = TRANSFER( tampon(2:4) , 1.0_8 , 3 )
y(:) = TRANSFER( tampon(2:4) , y(:) )
```

remplace la version Fortran 77 classique avec EQUIVALENCE [1] :

```
integer*8    tampon(4)
character*8  str,ch
real*8       x(3),y(3)
EQUIVALENCE (tampon(1),str) , (tampon(2),x)
ch = str
y(:) = x(:)
```

1. l'EQUIVALENCE est déclarée obsolète par la norme 2018.

Utilisation de la fonction TRANSFER pour passer une chaîne à C en évitant la transmission automatique de sa longueur.

```
integer,dimension(1) :: itab=0
character(len=10)    :: chain="0123456789"
call sub( transfer(chain//achar(0),itab) )
```

Conversions entiers/caractères

- `CHAR(i,[kind])`
  ⇒ $i^{ième}$ caractère de la table standard (`ASCII/EBCDIC`) si `kind` absent, sinon de la table correspondant à `kind` (*constructeur dépendant*).
- `ACHAR(i)`
  ⇒ idem `CHAR` avec table ASCII.
- `ICHAR(c,[kind])`
  ⇒ rang (entier) du caractère `c` dans la table associée à la valeur du mot-clé `kind` (`ASCII/EBCDIC en général`).
- `IACHAR(c)`
  idem `ICHAR` dans la table `ASCII`.

Comparaison de chaînes

- `LGE(string_a, string_b)`
  ⇒ `VRAI` si `string_a` « **supérieure ou =** » à `string_b`
  (*Lexically Greater or Equal*) ;
- `LGT(string_a, string_b)`
  ⇒ `VRAI` si `string_a` « **supérieure** » à `string_b`
  (*Lexically Greater Than*) ;
- `LLE(string_a, string_b)`
  ⇒ `VRAI` si `string_a` « **inférieure ou =** » à `string_b`
  (*Lexically Less or Equal*) ;
- `LLT(string_a, string_b)`
  ⇒ `VRAI` si `string_a` « **inférieure** » à `string_b`
  (*Lexically Less Than*).

Remarques

- La comparaison des chaînes s'effectue caractère par caractère à partir de la gauche en fonction de leur rang dans la table `ASCII` ;
- En cas d'inégalité de longueur, la chaîne la plus courte est complétée à blanc sur sa droite ;
- Ces quatre fonctions faisaient déjà partie de la norme 77 ;
- Les opérateurs >=, >, <= et < équivalents à ces fonctions peuvent aussi être utilisés. Il n'existe pas de fonctions `LEQ` et `LNE` équivalentes aux opérateurs == et /=.

Manipulation de chaînes

- `ADJUSTL(string)`
  ⇒ débarrasse `string` de ses blancs de tête (cadrage à gauche) et complète à droite par des blancs.
- `ADJUSTR(string)` ⇒ idem `ADJUSTL` mais à droite.
- `INDEX(string, substring [,back])`
  ⇒ numéro (entier) du premier caractère de `string` où apparaît la sous-chaîne `substring` (sinon 0). Si la variable logique `back` est *vraie* : recherche en sens inverse.
- `LEN_TRIM(string)`
  ⇒ longueur (entier) de la chaîne débarrassée de ses blancs de fin.
- `SCAN(string, set [,back])`
  ⇒ numéro (entier) du premier caractère de `string` figurant dans `set` ou 0 sinon. Si la variable logique `back` est *vraie* : recherche en sens inverse.
- `VERIFY(string, set [,back])`
  ⇒ numéro (entier) du premier caractère de `string` ne figurant pas dans `set`, ou 0 si tous les caractères de `string` figurent dans set. Si la variable logique `back` est *vraie* : recherche en sens inverse.
- `REPEAT(string, ncopies)`
  ⇒ chaîne obtenue en concaténant `ncopies` copies de `string`.
- `TRIM(string)` ⇒ débarrasse `string` de ses blancs de fin.

Opérations sur les bits

`IAND(i,j)`	fonction retournant un entier de même type que `i` résultant de la combinaison bit à bit de `i` et `j` par un `ET` **logique**.
`IEOR(i,j)`	fonction retournant un entier de même type que `i` résultant de la combinaison bit à bit de `i` et `j` par un `OU` **exclusif logique**.
`IOR(i,j)`	fonction retournant un entier de même type que `i` résultant de la combinaison bit à bit de `i` et `j` par un `OU` **inclusif logique**.
`ISHFT(i,shift)`	fonction retournant un entier de même type que `i` résultant du **décalage** de `shift` bits appliqué à `i`. Décalage vers la gauche ou vers la droite suivant que l'entier `shift` est positif ou négatif. Les bits sortant sont perdus et le remplissage se fait par des zéros.
`ISHFTC(i,shift[,size])`	fonction retournant un entier de même type que `i` résultant d'un **décalage circulaire** de `shift` positions appliqué aux `size` bits de droite de `i`. Décalage vers la gauche ou vers la droite suivant que l'entier `shift` est positif ou négatif.

`IBCLR(i,pos)`	fonction retournant un entier identique à `i` avec le `pos`$^{\text{ième}}$ bit mis à zéro ;
`IBSET(i,pos)`	fonction retournant un entier identique à `i` avec le `pos`$^{\text{ième}}$ bit mis à 1 ;
`NOT(i)`	fonction retournant un entier de même type que `i`, ses bits correspondant au **complément logique** de ceux de `i` ;
`IBITS(i,pos,len)`	fonction stockant dans un entier de même type que `i` les `len` bits de `i` à partir de la position `pos`. Ces bits sont cadrés à droite et complétés à gauche par des zéros ;
`MVBITS(from,frompos,len,to,topos)`	sous-programme copiant une séquence de bits depuis une variable entière (`from`) vers une autre (`to`).

`Remarques`

1. ces fonctions ont été étendues pour s'appliquer aussi à des tableaux d'entiers ;
2. `Norme 95` : le sous-programme `MVBITS` est « `pure` » et « `elemental` ».

paramètre KIND sur IBM/SP6

Type	Variante	Nb octets	Étendue	Précision
Entier	kind=1	1	$-128 \leq i \leq 127$	////////
	kind=2	2	$-2^{15} \leq i \leq 2^{15}-1$	////////
	kind=4	4	$-2^{31} \leq i \leq 2^{31}-1$	////////
	kind=8	8	$-2^{63} \leq i \leq 2^{63}-1$	////////
Réel	kind=4	4	$1.2 \times 10^{-38} \leq \mid r \mid \leq 3.4 \times 10^{38}$	6 chiffres
	kind=8	8	$2.2 \times 10^{-308} \leq \mid r \mid \leq 1.8 \times 10^{308}$	15 chiffres
	kind=16	16	$2.2 \times 10^{-308} \leq \mid r \mid \leq 1.8 \times 10^{308}$	31 chiffres
Complexe	kind=4	(4,4)	$1.2 \times 10^{-38} \leq \mid r \mid \leq 3.4 \times 10^{38}$	6 chiffres
	kind=8	(8,8)	$2.2 \times 10^{-308} \leq \mid r \mid \leq 1.8 \times 10^{308}$	15 chiffres
	kind=16	(16,16)	$2.2 \times 10^{-308} \leq \mid r \mid \leq 1.8 \times 10^{308}$	31 chiffres
Logique	kind=1	1	.true.=01, .false=00	////////
	kind=2	2	.true.=0001, .false=0000	////////
	kind=4	4	.true.=0..1, .false=0..0	////////
	kind=8	8	.true.=0..1, .false=0..0	////////
Caractère	kind=1	1	jeu ASCII	////////

paramètre KIND sur NEC/SX8

Type	Variante	Nb octets	Étendue	Précision
Entier	kind=2	2	$-2^{15} \leq i \leq 2^{15}-1$	////////
	kind=4	4	$-2^{31} \leq i \leq 2^{31}-1$	////////
	kind=8	8	$-2^{63} \leq i \leq 2^{63}-1$	////////
Réel	kind=4	4	$1.2 \times 10^{-38} \leq \mid r \mid \leq 3.4 \times 10^{38}$	6 chiffres
	kind=8	8	$2.2 \times 10^{-308} \leq \mid r \mid \leq 1.8 \times 10^{308}$	15 chiffres
	kind=16	16	$2.2 \times 10^{-308} \leq \mid r \mid \leq 1.8 \times 10^{308}$	31 chiffres
Complexe	kind=4	(4,4)	$1.2 \times 10^{-38} \leq \mid r \mid \leq 3.4 \times 10^{38}$	6 chiffres
	kind=8	(8,8)	$2.2 \times 10^{-308} \leq \mid r \mid \leq 1.8 \times 10^{308}$	15 chiffres
	kind=16	(16,16)	$2.2 \times 10^{-308} \leq \mid r \mid \leq 1.8 \times 10^{308}$	31 chiffres
Logique	kind=1	1	.true.=01, .false=00	////////
	kind=4	4	.true.=0..1, .false=0..0	////////
	kind=8	8	.true.=0..1, .false=0..0	////////
Caractère	kind=1	1	jeu caractères ASCII	////////
	kind=2	2	jeu caractères Japonais	////////

Exercice 1

Compiler et exécuter le programme contenu dans les fichiers `exo1.f90`, `mod1_exo1.f90` et `mod2_exo1.f90`. Recommencez en plaçant les modules dans un répertoire différent de celui où se trouve le programme principal.

`exo1.f90`

```
program exo1
  use mod1
  use mod2
  implicit none
  real    :: somme
  integer :: i

  tab = (/ (i*10,i=1,5) /)
  print *, tab
  call sp1s(somme)
  print *,somme
  call sp2s(somme)
  print *,somme
end program exo1
```

`mod1_exo1.f90`

```
module mod1
  real,dimension(5) :: tab
end module mod1
```

`mod2_exo1.f90`

```
module mod2
  use mod1
contains
  subroutine sp1s(som)
    implicit none
    real    som
    integer i

    som = 0.
    do i=1,5
      som = som + tab(i)
    enddo
  end subroutine sp1s
  !------------------
  subroutine sp2s(x)
    implicit none
    real x

    x = -x
  end subroutine sp2s
end module mod2
```

Exercice 2

Écrire une fonction récursive retournant la « `factorielle` » du nombre passé en argument. On pourra saisir ce nombre au clavier et permettre de multiples saisies.

Exercice 3

Écrire un programme permettant de valoriser la matrice *identité* de **n** lignes et **n** colonnes en évitant les traitements élémentaires via les boucles DO pour utiliser autant que possible les fonctions intrinsèques de manipulation de tableaux. Imprimer la matrice obtenue ligne par ligne et explorer plusieurs solutions mettant en œuvre les fonctions RESHAPE, UNPACK, CSHIFT ainsi que le bloc WHERE.

Exercice 4

Complétez le programme suivant en réécrivant les boucles imbriquées à l'aide d'une expression tableaux :

```
program exo4
  implicit none
  integer, parameter         :: nx=100, ny=200, nz=5
  real, dimension(nx,ny,nz)  :: tab1, tab2
  real                       :: vec1(nx), vec2(ny)
  integer                    :: i, j, k

  call random_number(vec1); call random_number(vec2)
  ! Calculs scalaires
  DO k=1,nz
    DO j=1,ny
      DO i=1,nx
        tab1(i,j,k)= sqrt(vec1(i))*exp(vec2(j))
      ENDDO
    ENDDO
  ENDDO

  ! Boucles précédentes réécrites à l'aide d'une expression tableaux :
  !                        ...
  !                        ...
  block
    integer, parameter :: k = 3
    logical            :: flag

    flag = MAXVAL(abs(tab1-tab2)) < MINVAL(k*spacing(tab1))
    print *, MERGE("tab1 et tab2 sont identiques", "tab1 et tab2 sont différents", flag)
  end block
end program exo4
```

Exercice 5

Écrire un programme permettant de valoriser une matrice de **n** lignes et **m** colonnes (**n** et **m** n'étant connus qu'au moment de l'exécution) de la façon suivante :

1. les lignes de rang pair seront constituées de l'entier 1,
2. les lignes de rang impair seront constituées des entiers successifs 1, 2, 3, ...

$$\text{Par exemple : } \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 1 & 1 & 1 \\ 5 & 6 & 7 & 8 \end{pmatrix}$$

Imprimer la matrice obtenue ligne par ligne afin de vérifier son contenu.

Exercice 6

Allouer une matrice réelle NxN (N multiple de 4) ; l'initialiser avec `real(i)` pour `i` variant de 1 à N*N. Transformer cette matrice en réordonnant les lignes et les colonnes de la façon suivante (pour N=16) :

```
 | 1  2| 3  4| 5  6| 7  8| 9 10|11 12|13 14|15 16|
```

transformé en :

```
 |15 16| 1  2|13 14| 3  4|11 12| 5  6| 9 10| 7  8|
```

Autrement dit, ramener les 2 dernières colonnes/lignes devant les 2 premières colonnes/lignes et garder ces 4 colonnes/lignes ensembles. Réitèrer ce processus en repartant des 2 dernières colonnes/lignes sans déplacer celles déjà transformées et ainsi de suite... Imprimer la matrice avant et après transformation et vérifier que la trace de la matrice est inchangée.

Exercice 7

Écrire un programme permettant de reconnaître si une chaîne est un *palindrome*. Lire cette chaîne dans une variable de type `character(len=long)` qui sera ensuite transférée dans un tableau (vecteur) de type `character(len=1)` pour faciliter sa manipulation via les fonctions intrinsèques tableaux. Écrire ce programme de façon modulaire et évolutive ; dans un premier temps, se contenter de lire la chaîne (simple mot) au clavier et dans un deuxième, ajouter la possibilité de lire un fichier (cf. fichier `palindrome`) contenant des phrases (suite de mots séparés par des blancs qu'il faudra supprimer – phase de *compression*).

Exercice 8

Compléter le programme contenu dans le fichier `exo8.f90` jusqu'à ce qu'il s'exécute correctement : les 2 matrices affichées devront être identiques.

exo8.f90

```
program exo8
  implicit none
  integer, parameter       :: n=5,m=6
  integer(kind=2)          :: i
  integer, dimension(0:n-1,0:m-1) :: a = &
       reshape((/ (i*100,i=1,n*m) /), shape(a))

  print *,"Matrice a avant appel à sp :"
  print *,"--------------------------"
  do i=0,size(a,1)-1
    print *,a(i,:)
  enddo
  call sp(a)
end program exo8
```

exo8.f90 (suite)

```
subroutine sp(a)
  integer                     :: i
  integer, dimension(:,:) :: a

  print *
  print *,"Matrice a dans sp :"
  print *,"-----------------"
  do i=0,size(a,1)-1
    print *,a(i,:)
  enddo
end subroutine sp
```

Exercice 9

Écrire un programme affichant les **n** premières lignes du triangle de Pascal avec allocation dynamique du triangle considéré comme un vecteur de lignes de longueur variable.

Par exemple :
$$\begin{pmatrix} 1 & & & & \\ 1 & 1 & & & \\ 1 & 2 & 1 & & \\ 1 & 3 & 3 & 1 & \\ 1 & 4 & 6 & 4 & 1 \end{pmatrix}$$

Exercice 10

Développez le module « surcharge » afin que le source suivant produise la sortie qui suit :

```
program exo10
  use surcharge

  integer i, j
  logical b1, b2

  b1 = .true.; b2 = .false.
  i = b1; j = b2
  print *,"b1 =", b1, " i =", i
  print *,"b2 =", b2, " j =", j

  if (b1 == b2) print *, "b1 == b2"
  if (b1 /= b2) print *, "b1 /= b2"
end program exo10
```

```
 Sortie :

 b1 = T, i = 1
 b2 = F, j = 0
 b1 /= b2
```

Exercice 11

Cet exercice reprend le module `matrix` des chapitres 10 et 11 (cf. page 219) du support de cours. Il est stocké (avec un programme principal d'utilisation complet) dans le fichier `exo11.f90`.

Complétez le module `matrix` en définissant un opérateur `.vp.` permettant le calcul des valeurs propres d'un objet de type `OBJ_MAT`. Utilisez par exemple le sous-programme `GEEV` (calculant les valeurs propres d'une matrice réelle d'ordre `N`) de la bibliothèque `LAPACK` dont voici le prototype :

```
SUBROUTINE GEEV( A, <w>, VL=vl, VR=vr, INFO=info )
     <type>(<wp>), INTENT(INOUT) :: A(:,:)
     <type>(<wp>), INTENT(OUT) :: <w(:)>
     <type>(<wp>), INTENT(OUT), OPTIONAL :: VL(:,:), VR(:,:)
     INTEGER, INTENT(OUT), OPTIONAL :: INFO
où
     <type> ::= REAL | COMPLEX
     <wp>   ::= KIND(1.0) | KIND(1.0D0)
     <w>    ::= WR, WI | W
     <w(:)> ::= WR(:), WI(:) | W(:)


A      (entrée/sortie) matrice carrée réelle ou complexe dont on désire
       les valeurs et/ou vecteur propres. En sortie son contenu est détruit.

<w>    (sortie) vecteur(s) réels ou complexe dans lesquels seront stockées les
       valeurs propres.
       Dans le cas réel, on précise deux vecteurs WR et WI dans lesquels
       seront stockées respectivement les parties réelles et imaginaires
       des valeurs propres. Une valeur propre et sa conjuguée sont stockées
       de façon consécutive, celle ayant la partie imaginaire positive avant
       l'autre.
       Dans le cas complexe, un seul vecteur W de type complexe est nécessaire,
       lequel est valorisé à l'aide des valeurs propres.
```

Exercice 11 (suite)

En entrée :

le programme principal lit (en « format libre » et avec le paramètre `ADVANCE="NO"`) un fichier `exo11.data` avec un enregistrement contenant :

- un entier représentant l'ordre `N` de la matrice,
- `N*N` valeurs réelles représentant les éléments de la matrice à traiter.

Un exemple d'un tel fichier (avec une matrice d'ordre `N=4`) est contenu dans `exo11.data`. Les valeurs propres attendues sont les suivantes : `(4.,0.)`, `(3.,0.)`, `(2.,0.)`, `(1.,0.)`.

Note :

- la *méthode* `imp_vp` a déjà été ajoutée au module `matrix` pour effectuer l'impression des valeurs propres.

Exercice 12

Soit le programme principal contenu dans le fichier `exo12.f90` :

```
program exo12
 use music
 type(musicien) :: mus_mort_le_plus_jeune

 call init
 call tri(critere="nom")
 call tri(critere="annee")
 mus_mort_le_plus_jeune = .MortLePlusJeune.tab_mus
 print *
 print *,"Le musicien mort le plus jeune est : ", nom(mus_mort_le_plus_jeune)
end program exo12
```

Dans le module `music` à créer, définir :

- le type `musicien` et un tableau `tab_mus` de ce type (dimensionné à 30).
- le sous-programme `init` devant lire le contenu du fichier `musiciens` (ce fichier contient une liste de compositeurs avec leurs années de naissance et de mort : éditez-le au préalable afin de connaître son formatage) afin de valoriser le tableau `tab_mus` et l'afficher,
- le sous-programme `tri` qui trie et affiche la liste des musiciens. Passer en argument le critère de tri sous forme d'une chaîne de caractères et effectuer ce tri par l'intermédiaire d'un tableau de pointeurs,

de sorte que l'exécution de ce programme produise les résultats suivants :


```
 ---- Liste des musiciens ----

     Johann Sebastian Bach                1685 1750
     Georg Friedrich  Haendel             1685 1759
     Wolfgang Amadeus Mozart              1756 1791
     Giuseppe         Verdi               1813 1901
     Richard          Wagner              1813 1883
     Ludwig van       Beethoven           1770 1827
              . . . . . .
              . . . . . .
     Igor             Stravinski          1882 1971
     Piotr Ilyitch    Tchaikovski         1840 1893
     Antonio          Vivaldi             1678 1741
     Carl Maria von   Weber               1786 1826
     Giacomo          Puccini             1858 1924
     Claude           Debussy             1862 1918
     Joseph           Haydn               1732 1809
     Gustav           Mahler              1860 1911


 ---- Liste alphabétique des musiciens ----

     Johann Sebastian Bach                1685 1750
     Ludwig van       Beethoven           1770 1827
     Johannes         Brahms              1833 1897
     Frederic         Chopin              1810 1849
     Claude           Debussy             1862 1918
     Georg Friedrich  Haendel             1685 1759
              . . . . . .
              . . . . . .
     Antonio          Vivaldi             1678 1741
     Richard          Wagner              1813 1883
     Carl Maria von   Weber               1786 1826
```

```
 ---- Liste chronologique des musiciens ----

     Claudio          Monteverdi           1567 1643
     Henry            Purcell              1659 1695
     Antonio          Vivaldi              1678 1741
     Johann Sebastian Bach                 1685 1750
     Georg Friedrich  Haendel              1685 1759
     Domenico         Scarlatti            1695 1757
              . . . . . .
              . . . . . .
     Maurice          Ravel                1875 1937
     Igor             Stravinski           1882 1971

Le musicien mort le plus jeune est : Gian-Battista Pergolese
```

Exercice 13

Même exercice que précédemment mais avec utilisation d'une *liste chaînée* simple ou double en partant du programme principal suivant contenu dans le fichier exo13.f90 :

```
program exo13
  use music
  type(musicien) :: mus_mort_le_plus_jeune

  call init
  call tri(critere="nom")
  call tri(critere="annee")
  mus_mort_le_plus_jeune = .MortLePlusJeune.debut
  print *
  print *,"Le musicien mort le plus jeune est : ", nom(mus_mort_le_plus_jeune)
end program exo13
```

Remarque : debut correspond au pointeur de début de liste.

Corrigé de l'exercice 1 à l'aide du compilateur d'INTEL (l'utilitaire "makedepf90" permet de générer les dépendances)

Corrigé de l'exercice 1

```
$ ifort -c mod1_exo1.f90
$ ifort -c mod2_exo1.f90
$ ifort exo1.f90 mod1_exo1.o mod2_exo1.o -o exo1
$ ./exo1
```

Remarque : si les modules sont situés dans un répertoire **rep1** différent de celui (**rep2**) d'exo1, utiliser l'option **-I** :

```
$ cd $HOME/rep1
$ ifort -c mod1_exo1.f90 mod2_exo1.f90
$ cd ../rep2
$ ifort exo1.f90 -I../rep1 ../rep1/mod*.o -o exo1
$ ./exo1
```

Exemple à l'aide d'un fichier "makefile" :

```
$ cat makefile
SRCS= mod1_exo1.f90 mod2_exo1.f90 exo1.f90
OBJS= $(SRCS:f90=o)
FC=ifort
FFLAGS=
.SUFFIXES: .f90

all: exo1

.f90.o:
        $(FC) $(FFLAGS) -c $<

clean:
        rm -f $(OBJS) exo1

exo1: $(OBJS)
        $(FC) $(OBJS) -o $@
$ makedepf90 mod1_exo1.f90 mod2_exo1.f90 exo1.f90 >> makefile ; make
$ makedepf90 mod1_exo1.f90 mod2_exo1.f90 exo1.f90 -o exo1 > makefile ; make
```

Corrigé de l'exercice 2 (solution sans module)

```
program exo2
  use ISO_FORTRAN_ENV, only : INPUT_UNIT
  implicit none
  integer ios, n
  integer, external :: f

  do
    print *,"Entrez une valeur :"
    read(unit=INPUT_UNIT, fmt=*, iostat=ios) n
    if (ios == 0) then
      print '(2(a,i0))', &
          "factorielle(", n, ") = ", f(n)
      cycle
    end if
    if (ios < 0) exit
    if (ios > 0) print *,"saisie invalide !"
  end do
end program exo2
```

Corrigé de l'exercice 2 (solution sans module)

```
integer recursive function f(n) &
        result(fact)
  integer, intent(in) :: n

  if (n == 1) then
    fact = 1
  else
    fact = n*f(n-1)
  endif
end function f
```

Corrigé de l'exercice 2 (solution avec module)

```
module factorielle
  implicit none
contains
  integer recursive function f(n) &
          result(fact)
    integer, intent(in) :: n

    if (n == 1) then
      fact = 1
    else
      fact = n*f(n-1)
    endif
  end function f
end module factorielle
```

Corrigé de l'exercice 2 (solution avec module)

```
program exo2
  use ISO_FORTRAN_ENV, only : INPUT_UNIT
  use factorielle
  implicit none
  integer ios, n

  do
    print *,"Entrez une valeur :"
    read(unit=INPUT_UNIT, fmt=*, iostat=ios) n
    if (ios == 0) then
      print '(2(a,i0))', &
          "factorielle(", n, ") = ", f(n)
      cycle
    end if
    if (ios < 0) exit
    if (ios > 0) print *,"saisie invalide !"
  end do
end program exo2
```

Corrigé de l'exercice 3

```
program exo3
  implicit none
  integer, parameter       :: n = 10
  real   , dimension(n,n) :: mat_ident
  character(len=8)         :: fmt = "(00f3.0)"

  write(fmt(2:3), "(i2)") n  ! Format d'impression
  !==============> Première solution :
  call sol_unpack
  call imp
  !==============> Deuxième solution :
  call sol_reshape
  call imp
  !==============> Troisième solution :
  call sol_cshift
  call imp

  contains
    subroutine sol_unpack
      logical, dimension(n,n) :: mask
      real,    dimension(n)   :: diag = 1.
      integer                 :: i, j

      mask = reshape(source=[ ((i==j, i=1,n), j=1,n) ], shape=shape(mask))
      mat_ident = unpack(diag, mask, 0.)
    end subroutine sol_unpack
```

Corrigé de l'exercice 3 (suite)

```
    subroutine sol_reshape
      real, dimension(n*n) :: vect = 0.

      vect(1:n*n:n+1) = 1.
      mat_ident = reshape( vect, shape=shape(mat_ident) )
    end subroutine sol_reshape
    !
    subroutine sol_cshift
      integer i

      mat_ident(:,:) = 0.
      mat_ident(:,1) = 1.
      mat_ident(:,:) = cshift( array=mat_ident, &
                               shift=(/ (-i,i=0,n-1) /), &
                               dim=2 )
    end subroutine sol_cshift
    !
    subroutine imp
      integer i

      do i=1,n   ! Impression matrice identité
        print fmt, mat_ident(i,:)
      end do
      print *
    end subroutine imp
end program exo3
```

Corrigé de l'exercice 4

```
program exo4
  implicit none
  integer, parameter         :: nx=100, ny=200, nz=5
  real, dimension(nx,ny,nz)  :: tab1, tab2
  real, dimension(nx)        :: vec1
  real, dimension(ny)        :: vec2
  integer i, j, k

  call random_number(vec1) ; call random_number(vec2)
  ! Calculs scalaires
  DO k=1,nz
    DO j=1,ny
      DO i=1,nx
        tab1(i,j,k)= sqrt(vec1(i))*exp(vec2(j))
      ENDDO
    ENDDO
  ENDDO
  ! Boucles précédentes réécrites :
  tab2(:,:,:) = SPREAD(source=SPREAD(source=sqrt(vec1(:)),dim=2,ncopies=ny)* &
                               SPREAD(source=exp(vec2(:)) ,dim=1,ncopies=nx), &
                       dim=3,ncopies=nz)
  !tab2(:,:,:) = &
  !   SPREAD(source=reshape(source=[ ((sqrt(vec1(i))*exp(vec2(j)),i=1,nx),j=1,ny) ], &
  !                         shape=[nx,ny]), dim=3, ncopies=nz)
  block
    integer, parameter :: k = 3
    logical            :: flag
    flag = MAXVAL(abs(tab1-tab2)) < MINVAL(k*spacing(tab1))
    print *, MERGE("tab1 et tab2 sont identiques", "tab1 et tab2 sont différents", flag)
  end block
end program exo4
```


Corrigé de l'exercice 5

```
program exo5
  implicit none
  ! On décide que les entiers « n,m » sont < 100
  integer, parameter                    :: p = selected_int_kind(2)
  integer(kind=p)                       :: n,m
  integer                               :: i, j
  integer, dimension(:,:), allocatable  :: mat
  character(len=6)                      :: fmt = "(  I6)"
  !
  print *,"Nombre de lignes :"  ; read(*,*)n
  print *,"Nombre de colonnes :"; read(*,*)m
  write(fmt(2:3), '(i2)') m
  allocate(mat(n,m))
  ! Remplissage des lignes paires avec l'entier 1
  mat(2::2,:) = 1
  ! Remplissage lignes impaires avec les entiers 1,2,...
  mat(1::2,:) = reshape( source=(/ (i,i=1,size(mat(1::2,:))) /), &
                         shape=shape(mat(1::2,:)),               &
                         order=(/ 2,1 /) )
  print fmt, ((mat(i,j),j=1,m),i=1,n)
  deallocate(mat)
end program exo5
```

Corrigé de l'exercice 6

```
program exo6
  implicit none
  real,    dimension(:,:), allocatable :: A
  logical, dimension(:,:), allocatable :: m
  integer                              :: N, i, j
  character(len=8)                     :: fmt = "(  F5.0)"

  !------- Allocation/initialisation de A --------------
  print *, "N (multiple de 4 < 32) ?" ; read *, N
  allocate(A(N,N), m(N,N))
  write(fmt(2:3), '(i2)') N
  A = reshape(source=[ (real(i), i=1,N*N) ],   shape=shape(A), order=[ 2,1 ])
  m = reshape(source=[ ((i==j,i=1,N),j=1,N) ], shape=shape(m))
  call imp("Matrice à transformer :")
  !------------- Transformation des lignes -------------
  do i=1,N,4
    A(:,i:N) = cshift(array=A(:,i:N), shift=-2, dim=2)
  end do
  !------------- Transformation des colonnes -----------
  do i=1,N,4
    A(i:N,:) = cshift(array=A(i:N,:), shift=-2, dim=1)
  end do
  call imp("Matrice transformée :")
  deallocate(A) ; deallocate(m)
  contains
    subroutine imp(titre)
      character(len=*) titre

      print "(/,A,/)", titre
      print fmt, ((A(i,j),j=1,N),i=1,N)
      print *, "trace = ", sum(pack(array=A, mask=m))
    end subroutine imp
end program exo6
```


Corrigé de l'exercice 7

```
program exo7
  integer, parameter  :: long=80
  character(len=long) :: chaine
  integer             :: long_util, ios, choix
  logical             :: entree_valide

  do
    entree_valide = .true.
    print *, "1) Entrée clavier"
    print *, "2) Lecture fichier 'palindrome'"
    read(unit=*, fmt=*, iostat=ios) choix
    if(ios > 0) entree_valide = .false.; if(ios < 0) stop "Arrêt demandé"
    if(choix /= 1 .and. choix /= 2) entree_valide=.false.
    if(entree_valide ) exit
    print *, "Entrée invalide"
  end do
  if (choix==2) open(unit=1, file="palindrome", form="formatted", action="read")
  do
    select case(choix)
      case(1)
        print *, "Entrez une chaîne :" ; read(unit=*, fmt="(a)", iostat=ios) chaine
      case(2)
        read(unit=1, fmt="(a)", iostat=ios) chaine
    end select
    if(ios > 0) stop "Erreur de lecture"; if(ios < 0) exit
    long_util = len_trim(chaine)
    block
      character(len=10) str
      str = MERGE(TSOURCE=" est      ", FSOURCE=" n'est pas", &
                  MASK=palind( chaine(:long_util)))
      print *, chaine(:long_util), trim(str), " un palindrome"
    end block
  enddo
  if (choix == 2) close(unit=1)
```

Corrigé de l'exercice 7 (suite)

```
contains
  function palind(chaine)
    logical          :: palind  !<-- Retour fonction
    character(len=*) :: chaine  !<-- Argument muet
    ! Déclarations de variables locales
    character(len=1), dimension(len(chaine)) :: tab_car   !<-- Tableau automatique
    integer                                  :: long_util

    ! Copie chaîne entrée dans un tableau de caractères.
    tab_car(:) = transfer(chaine, "a", size(tab_car))
    ! Dans le cas où la chaîne contient une phrase,
    ! on supprime les blancs séparant les différents mots.
    long_util = compression(tab_car(:))
    ! Comparaison des éléments symétriques par rapport
    ! au milieu de la chaîne. La fonction « all » nous sert
    ! à comparer le contenu de deux tableaux.
    palind=all(tab_car(:long_util/2) == tab_car(long_util:long_util-long_util/2+1:-1))
  end function palind

  function compression(tab_car) result(long_util)
    integer :: long_util !<-- Retour fonction
    character(len=1), dimension(:)             :: tab_car !<-- Profil implicite
    logical,          dimension(size(tab_car)) :: m       !<-- Tableau automatique

    m(:) = tab_car(:) /= " "
    long_util = count(mask=m(:))
    tab_car(:long_util) = pack(array=tab_car(:), mask=m(:))
  end function compression
end program exo7
```


Corrigé de l'exercice 7 : version norme 2003

```
module exo7_m
  implicit none
  integer :: choix
contains
  subroutine choix_fichier
    logical :: entree_valide
    integer :: ios

    do
      entree_valide = .true.
      print *, '1) Entrée clavier'
      print *, '2) Lecture fichier "palindrome"'
      print *, ' Faire CtrD pour quitter'
      read(unit=*, fmt=*, iostat=ios) choix
      if (ios > 0) entree_valide = .false.
      if (ios < 0) stop "Arrêt demandé"
      if (choix /= 1 .and. choix /= 2) entree_valide=.false.
      if (entree_valide) exit
      print *, "Entrée invalide"
    end do

    if (choix == 2) open(unit=1, file="palindrome", &
                         form="formatted", action="read")
  end subroutine choix_fichier
```

Corrigé de l'exercice 7 : version norme 2003 (suite)

```
subroutine traitement_fichier
  integer, parameter  :: long=80
  character(len=long) :: chaine
  integer             :: ios, long_util

  do
    select case(choix)
      case(1)
        print *, "Entrez une chaîne :"
        read(unit=*, fmt='(a)', iostat=ios) chaine
      case(2)
        read(unit=1, fmt='(a)', iostat=ios) chaine
    end select
    if(ios > 0) stop "Erreur de lecture"
    if(ios < 0) exit
    !
    ! Récup. longueur chaîne entrée (sans blancs de fin).
    !
    long_util = len_trim(chaine)
    block
      character(len=10) str

      str = MERGE(TSOURCE=" est      ", FSOURCE=" n'est pas", &
                  MASK=palind( chaine(:long_util)))
      print *, chaine(:long_util), trim(str), " un palindrome"
    end block
  enddo
  if (choix == 2) close(unit=1)
end subroutine traitement_fichier
```


Corrigé de l'exercice 7 : version norme 2003 (suite)

```
function palind(chaine)
  ! Déclaration argument et retour de fonction
  logical                                    :: palind
  character(len=*)                           :: chaine
  ! Déclarations locales
  character(len=1), dimension(:), allocatable :: tab_car
  integer                                    :: long_util, i

  ! Copie chaîne entrée dans un tableau de caractères.
  ! Norme 2003 : l'instruction qui suit alloue[1] le tableau
  ! ----------   « tab_car » avec la taille nécessaire afin
  !              de respecter la conformance.
  tab_car = [ (chaine(i:i), i=1, len(chaine)) ]
  !
  ! Dans le cas où la chaîne contient une phrase,
  ! on supprime les blancs séparant les différents mots.
  !
  call compression(tab_car)
  !
  ! Comparaison des éléments symétriques par rapport
  ! au milieu de la chaîne. La fonction « all » nous sert
  ! à comparer le contenu de deux tableaux.
  !
  long_util = size(tab_car(:))
  palind = all( tab_car(:long_util/2) == &
                tab_car(long_util:long_util-long_util/2+1:-1) )
```

1. Attention : avec le compilateur « *ifort* » d'INTEL il peut être nécessaire, suivant le niveau de version, de positionner l'option « *-assume realloc_lhs* » pour bénéficier de ce mécanisme d'allocation/réallocation automatique.

Corrigé de l'exercice 7 : version norme 2003 (suite)

```
  contains

    subroutine compression(tab_car)
      character(len=1), dimension(:), allocatable :: tab_car ! norme 2003
      logical,          dimension(size(tab_car))  :: m     !<-- Tabl. automatique

      m(:) = tab_car(:) /= ' '
      ! Norme 2003 : l'instruction suivante réalloue de façon automatique[1]
      ! ----------   le tableau « tab_car » afin de respecter la conformance.
      tab_car = pack(array=tab_car(:), mask=m(:))
    end subroutine compression
  end function palind
end module exo7_m

program exo7
  use exo7_m

  call choix_fichier
  call traitement_fichier
end program exo7
```

1. Attention : avec le compilateur « *ifort* » d'INTEL il peut être nécessaire, suivant le niveau de version, de positionner l'option « *-assume realloc_lhs* » pour bénéficier de ce mécanisme d'allocation/réallocation automatique.

Corrigé de l'exercice 8

```
program exo8
  ! Dans « sp » l'argument « a » est un tableau à profil implicite, l'interface doit donc
  ! être explicite, d'où l'ajout du « bloc interface ». Dans ce contexte, seul le profil de
  ! « a » est transmis, pas les bornes inférieures qui sont considérées égales à 1.
  implicit none
  interface
    subroutine sp(a)
      integer, dimension(:,:) :: a
    end subroutine sp
  end interface
  integer, parameter                :: n=5,m=6
  integer(kind=2)                   :: i
  integer, dimension(0:n-1,0:m-1) :: a = reshape((/ (i*100,i=1,n*m) /), shape(a))

  print *, "Matrice a avant appel à sp :"
  print *, "---------------------------"
  print *
  do i=lbound(a,1),ubound(a,1)
    print *,a(i,:)
  enddo
  call sp(a)
end program exo8
!------------------
subroutine sp(a)
  integer, dimension(:,:) :: a ! ou « integer, dimension(0:,0:) :: a » (bornes inf. à 0)
  integer                 :: i

  print *
  print *, "Matrice a dans sp :"
  print *, "------------------"
  print *
  do i=lbound(a,1),ubound(a,1)
    print *,a(i,:)
  enddo
end subroutine sp
```

Corrigé de l'exercice 9 (solution Fortran 2003)

```
program exo9
  implicit none
  type ligne
    integer, dimension(:), allocatable :: p ! à la place de « pointer » : norme 2003
  end type ligne
  type(ligne), dimension(:), allocatable :: triangle
  integer i, n, etat

  do
    write(*, advance="no", fmt="('Ordre du triangle ? :')")
    read(*, *, iostat=etat)n
    if (etat /= 0 .OR. n > 20) exit ! On limite à 20 lignes
    allocate(triangle(n))
    do i=1,n
      !--- Pour chaque ligne du triangle, allocation du nombre de colonnes.
      allocate(triangle(i)%p(i))
      !-Valorisation des éléments extrêmes de la ligne courante puis des
      !-éléments intermédiaires au moyen d'une expression vectorielle.
      triangle(i)%p([ 1,i ]) = 1
      !-Pour « i=2 », les opérandes de l'expression suivante sont
      !-des tableaux de taille nulle :  « zero-sized array »
      if (i>1) &
        triangle(i)%p(2:i-1) = triangle(i-1)%p(2:i-1) + &
                               triangle(i-1)%p(1:i-2)
      print "(*(i5,1x))",triangle(i)%p ! « * » : fortran 2008
    end do
    !-- Libération du triangle. Les composantes « p » de
    !-- chacun des éléments du triangle seront libérées
    !-- automatiquement.
    deallocate(triangle)
  end do
end program exo9
```

Corrigé de l'exercice 10

```
module surcharge
  interface assignment(=)
    module procedure affect_int_bool
  end interface
  interface operator(==)
    module procedure compare_eq_bool
  end interface
  interface operator(/=)
    module procedure compare_ne_bool
  end interface

contains

  logical function compare_eq_bool(l1, l2)
    logical, intent(in) :: l1, l2

    compare_eq_bool = l1.EQV.l2
  end function compare_eq_bool

  logical function compare_ne_bool(l1, l2)
    logical, intent(in) :: l1, l2

    compare_ne_bool = l1.NEQV.l2
  end function compare_ne_bool

  subroutine affect_int_bool(i, l)
    integer, intent(inout) :: i
    logical, intent(in)    :: l

    i = MERGE(tsource=1, fsource=0, mask=l)
  end subroutine affect_int_bool
end module surcharge
```

Corrigé de l'exercice 11

```
module matrix
  use f95_lapack, only: GEEV => LA_GEEV
  implicit none
  integer          :: nb_lignes, nb_col
  integer, private :: err

  type OBJ_MAT
    private
    integer                              :: n=0, m=0
    real, dimension(:,:), allocatable :: mat
  end type OBJ_MAT

  private  :: add, trans, taille_mat, valorisation, affect
  private  :: val_propres
  !-------------------------------
  interface operator(+)
     module procedure add
  end interface
  !-------------------------------
  interface operator(.tr.)
    module procedure trans
  end interface
  !-------------------------------!
  interface operator(.vp.)        !
     module procedure val_propres !
  end interface                   !
  !-------------------------------!
  interface assignment(=)
    module procedure taille_mat, valorisation, affect
  end interface
contains
```

Corrigé de l'exercice 11 (suite)

```
  subroutine valorisation(a,t)
    type(OBJ_MAT),       intent(inout) :: a
    real, dimension(:), intent(in)    :: t
     ....
  end subroutine valorisation
  subroutine poubelle(a)
    type(OBJ_MAT), intent(inout) :: a
     ....
  end subroutine poubelle
  function add(a,b)
    type(OBJ_MAT), intent(in) :: a,b
    type(OBJ_MAT)             :: add
     ....
  end function add
  function trans(a)
    type(OBJ_MAT), intent(in) :: a
    type(OBJ_MAT)             :: trans
     ....
  end function trans
  !-------------------------------------------------!
  function val_propres(a)                           !
    type(OBJ_MAT), intent(in)     :: a              !
    complex, dimension(nb_lignes) :: val_propres    !
    real,    dimension(nb_lignes)       :: Wr, Wi   !
    real,    dimension(:,:), allocatable :: matrice !
                                                     !
    if (allocated(a%mat)) then                       !
      matrice = a%mat                                !
      call GEEV(matrice, Wr, Wi)                     !
      val_propres = cmplx(Wr, Wi)                    !
    else                                             !
      print *, "Objet non existant"                  !
    end if                                           !
  end function val_propres                           !
  !-------------------------------------------------!
```

Corrigé de l'exercice 11 (suite)

```
  subroutine taille_mat(i,a)
    integer,        intent(out) :: i
    type(OBJ_MAT), intent(in)  :: a
    i = a%n*a%m
  end subroutine taille_mat
  subroutine affect(a,b)
    type(OBJ_MAT), intent(inout) :: a
    type(OBJ_MAT), intent(in)    :: b
    ....
  end subroutine affect
  subroutine imp(a)
    type(OBJ_MAT), intent(in)  :: a
    integer(kind=2)            :: i

    print "(//, a, /)", "              Matrice : "
    do i=1,a%n
      print '(*(f6.2))',a%mat(i,:) ! « * » dans format : Fortran 2008
    enddo
  end subroutine imp
  subroutine imp_vp(vec)
    complex, dimension(:) :: vec
    integer               :: i

    print "(//, a, /)", "            Valeurs propres : "
    do i=1,size(vec)
      print '("Valeur propre N.", i2, " : (", 1pe9.2, &
            "  , ", 1pe9.2, ")")', i, real(vec(i)), aimag(vec(i))
    end do
  end subroutine imp_vp
end module matrix
```

Corrigé de l'exercice 11 (suite)

```
program exo11
  use matrix
  type(OBJ_MAT)   :: u
  real,    dimension(:), allocatable :: val_init
  complex, dimension(:), allocatable :: val_pr

  open(unit=10, file="exo11.data", form="formatted", action="read")
  read(10, advance="no", fmt="(i1)") nb_lignes
  nb_col = nb_lignes
  allocate(val_init(nb_lignes*nb_col))
  read(10, *) val_init
  close(10)
  u = val_init
  deallocate(val_init)
  call imp(u)
  !---------------
  val_pr = .vp. u
  !---------------
  call imp_vp(val_pr)
  call poubelle(u)
end program exo11
```

Corrigé de l'exercice 12

```
module music
  implicit none
  integer, parameter :: nb_enr=30
  integer            :: nb_mus
  !-----------------------------------------------
  type musicien
    private
    character(len=16) :: prenom
    character(len=21) :: nom
    integer           :: annee_naiss,annee_mort
  end type musicien
  !-----------------------------------------------
  type , private :: ptr_musicien
    type(musicien), pointer :: ptr
  end type ptr_musicien
  !-------------------------------------------------------------------------
  type(musicien),     dimension(nb_enr), target          :: tab_mus
  type(ptr_musicien), dimension(:), allocatable, private :: tab_ptr_musicien
  !-------------------------------------------------------------------------
  interface operator(<)
    module procedure longevite
  end interface
  !-----------------------------------
  interface operator(.MortLePlusJeune.)
    module procedure mort_le_plus_jeune
  end interface
  !-----------------------------------
  private :: operator(<), longevite, mort_le_plus_jeune
```

Corrigé de l'exercice 12 (suite)

```
contains
  subroutine init
    integer :: etat,i
    ! Valorisation du tableau de musiciens
    open(1,file="musiciens",action="read",status="old")
    nb_mus = 0
    do
      read(1,"(a,1x,a,2(1x,i4))", iostat=etat) tab_mus(nb_mus+1)
      if (etat /= 0) exit
      nb_mus = nb_mus + 1
    enddo
    close(1)
    !
    ! On alloue le tableau de pointeurs dont le nombre
    ! d'éléments correspond au nombre de musiciens.
    !
    allocate(tab_ptr_musicien(nb_mus))
    !
    ! Chaque élément du tableau de pointeurs alloué précédemment va être mis
    ! en relation avec l'élément correspondant du tableau de musiciens.
    ! Chaque élément du tableau de musiciens « tab_mus » a l'attribut « target »
    ! implicitement car cet attribut a été spécifié pour le tableau lui-même.
    !
    do i=1,nb_mus
      tab_ptr_musicien(i)%ptr => tab_mus(i)
    enddo
    print *,"---- Liste des musiciens ----"
    print *
    write(*,"((5x,a,1x,a,2(1x,i4)))") (tab_mus(i),i=1,nb_mus)
  end subroutine init
```

Corrigé de l'exercice 12 (suite)

```
  subroutine tri(critere)
    ! Procédure triant la liste des musiciens par ordre alphabétique des noms
    ! ou par ordre chronologique en fonction du paramètre « critere » spécifié.
    ! Ce tri s'effectue par l'intermédiaire du tableau de pointeurs « tab_ptr_musicien ».
    ! Déclaration de l'argument
    character(len=*), intent(in) :: critere
    ! Déclarations locales
    character(len=*), parameter, dimension(2) :: labels = &
            [ "alphabetique ", "chronologique" ]
    character(len=len(labels))                :: label_tri
    logical                                   :: expr
    integer                                   :: i, j
    !
    do i=nb_mus-1,1,-1
      do j=1,i
        select case(critere)
          case("nom")
            label_tri = labels(1)
            expr = tab_ptr_musicien(j)%ptr%nom > &
                   tab_ptr_musicien(j+1)%ptr%nom
          case("annee")
            label_tri = labels(2)
            expr = tab_ptr_musicien(j)%ptr%annee_naiss > &
                   tab_ptr_musicien(j+1)%ptr%annee_naiss
        end select
        if (expr) &
          !---------Permutation des deux associations---------
          tab_ptr_musicien(j:j+1) = tab_ptr_musicien(j+1:j:-1)
          !---------------------------------------------------
      enddo
    enddo
    !
    print "(/, a, a, a, /)", "---- Liste ", trim(label_tri), " des musiciens ----"
    write(*,"((5x,a,1x,a,2(1x,i4)))") (tab_ptr_musicien(i)%ptr,i=1,nb_mus)
    end subroutine tri
```


Corrigé de l'exercice 12 (suite)

```
  function longevite(mus1,mus2)
    !-- Fonction surchargeant l'opérateur < afin de pouvoir spécifier
    !-- des opérandes de type « musicien ».
    type(musicien), intent(in) :: mus1,mus2
    logical                    :: longevite
    integer                    :: duree_de_vie_mus1, &
                                  duree_de_vie_mus2
    duree_de_vie_mus1 = mus1%annee_mort - mus1%annee_naiss
    duree_de_vie_mus2 = mus2%annee_mort - mus2%annee_naiss
    longevite = duree_de_vie_mus1 < duree_de_vie_mus2
  end function longevite
  !----------------------
  function mort_le_plus_jeune(tab_mus)
    ! procédure de définition de l'opérateur « .MortLePlusJeune. »
    type(musicien), dimension(:), intent(in) :: tab_mus
    type(musicien)                           :: mort_le_plus_jeune
    integer i

    mort_le_plus_jeune = tab_mus(1)
    do i=2,nb_mus
      ! Ici l'utilisation de l'opérateur < provoque l'appel à la fonction « longevite » :
      !    tab_mus(i) < mus <=> longevite(tab_mus(i),mus)

      if (tab_mus(i) < mort_le_plus_jeune) mort_le_plus_jeune = tab_mus(i)
    enddo
  end function mort_le_plus_jeune
  !--------------------------------
  function nom(mus)
    !-- Fonction renvoyant les nom et prénom du musicien passé en argument.
    type(musicien), intent(in) :: mus
    character(len=38)          :: nom

    nom = trim(mus%prenom)//" "//mus%nom
  end function nom
end module music
```

Corrigé de l'exercice 13 : solution avec liste chaînée simple

```
module music
  implicit none
  !
  type musicien
    private
    character(len=16)        :: prenom
    character(len=21)        :: nom
    integer                  :: annee_naiss,annee_mort
    type(musicien), pointer :: ptr
  end type musicien
  !---------------------------------
  type(musicien), pointer    :: debut
  !---------------------------------
  interface operator(<)
    module procedure longevite
  end interface
  !-------------------------------------
  interface operator(.MortLePlusJeune.)
    module procedure mort_le_plus_jeune
  end interface
  !-------------------------------------
  private :: operator(<), longevite, mort_le_plus_jeune, liste
```

Corrigé de l'exercice 13 : solution avec liste chaînée simple (suite)

```
contains
  subroutine init
    type(musicien)           :: mus
    type(musicien), pointer  :: ptr_precedent, ptr_courant
    integer                  :: etat

    nullify(debut)
    nullify(mus%ptr)
    open(1,file="musiciens",action="read",status="old")
    do
      read(1,"(a,1x,a,2(1x,i4))",iostat=etat) mus%prenom,      &
                                              mus%nom,         &
                                              mus%annee_naiss, &
                                              mus%annee_mort
      if (etat /= 0) exit
      allocate(ptr_courant)
      if (.not.associated(debut)) then
        debut => ptr_courant
      else
        ptr_precedent%ptr => ptr_courant
      endif
      ptr_precedent => ptr_courant
      ptr_courant = mus
    enddo
    close(1)
    print *
    print *,"---- Liste des musiciens ----"
    print *
    call liste
  end subroutine init
```

Corrigé de l'exercice 13 : solution avec liste chaînée simple (suite)

```
subroutine tri(critere)
  ! Procédure triant la liste des musiciens par ordre alphabétique des noms
  ! ou par ordre chronologique en fonction du paramètre « critere » spécifié.
  character(len=*), intent(in) :: critere
  ! Déclarations locales
  character(len=*), parameter, dimension(2) :: labels = &
          [ "alphabetique ", "chronologique" ]
  character(len=len(labels))                :: label_tri
  type(musicien), pointer                   :: ptr_courant, ptr_precedent, temp
  logical                                   :: tri_termine, expr
  do
    tri_termine = .true. ; ptr_courant => debut ; ptr_precedent => debut
    do
      if (.not.associated(ptr_courant%ptr)) exit
      select case(critere)
        case("nom")
          label_tri = labels(1) ; expr = ptr_courant%nom > ptr_courant%ptr%nom
        case("annee")
          label_tri = labels(2) ; expr = ptr_courant%annee_naiss > &
                                         ptr_courant%ptr%annee_naiss
      end select
      if (expr) then
        if (associated(ptr_courant, debut)) then
          debut => ptr_courant%ptr
        else
          ptr_precedent%ptr => ptr_courant%ptr
        end if
        ptr_precedent       => ptr_precedent%ptr ; temp => ptr_courant%ptr%ptr
        ptr_courant%ptr%ptr => ptr_courant       ; ptr_courant%ptr => temp
        tri_termine = .false. ; cycle
      end if
      ptr_precedent => ptr_courant
      if (associated(ptr_courant%ptr)) ptr_courant => ptr_courant%ptr
    end do
    if (tri_termine) exit
  end do
  print *; print *,"---- Liste ", trim(label_tri), " des musiciens ----"; print *
  call liste
end subroutine tri
```


Corrigé de l'exercice 13 : solution avec liste chaînée simple (suite)

```
function longevite(mus1,mus2)
  ! Fonction surchargeant l'opérateur < afin de pouvoir spécifier des
  ! opérandes de type « musicien ».
  type(musicien), intent(in) :: mus1,mus2
  logical                    :: longevite
  integer                    :: duree_de_vie_mus1, duree_de_vie_mus2

  duree_de_vie_mus1 = mus1%annee_mort - mus1%annee_naiss
  duree_de_vie_mus2 = mus2%annee_mort - mus2%annee_naiss
  longevite = duree_de_vie_mus1 < duree_de_vie_mus2
end function longevite
function mort_le_plus_jeune(debut)
  type(musicien), intent(in) :: debut
  type(musicien)             :: mort_le_plus_jeune
  type(musicien), pointer    :: p_mus

  mort_le_plus_jeune = debut; p_mus => debut%ptr
  do while(associated(p_mus))
    ! Ici l'utilisation de l'opérateur < provoque l'appel à la fonction « longevite » :
    !    p_mus < mort_le_plus_jeune <=> longevite(p_mus, mort_le_plus_jeune)

    if (p_mus < mort_le_plus_jeune) mort_le_plus_jeune = p_mus
   p_mus => p_mus%ptr
 enddo
end function mort_le_plus_jeune
!--------------------------------
function nom(mus)
  ! Fonction renvoyant les nom et prénom du musicien passé en argument.
  type(musicien), intent(in) :: mus
  character(len=38)          :: nom

  nom = trim(mus%prenom)//" "//mus%nom
end function nom
```

Corrigé de l'exercice 13 : solution avec liste chaînée simple (suite)

```
  subroutine liste
    type(musicien), pointer :: ptr_courant

    ptr_courant => debut
    if (.not.associated(debut)) then
      print *,"Il n'existe aucun musicien !"
      stop  8
    end if
    do
      write(*,"((5x,a,1x,a,2(1x,i4)))") ptr_courant%prenom,      &
                                        ptr_courant%nom,         &
                                        ptr_courant%annee_naiss, &
                                        ptr_courant%annee_mort
      if (.not.associated(ptr_courant%ptr)) exit
      ptr_courant => ptr_courant%ptr
    end do
  end subroutine liste
end module music
```

Corrigé de l'exercice 13 : solution avec liste chaînée double

```
module music
  implicit none
  !
  type musicien
    private
    character(len=16)        :: prenom
    character(len=21)        :: nom
    integer                  :: annee_naiss,annee_mort
    type(musicien), pointer  :: ptr_precedent, ptr_suivant
  end type musicien
  !---------------------------------
  type(musicien), pointer    :: debut
  !---------------------------------
  interface operator(<)
    module procedure longevite
  end interface
  !-----------------------------------
  interface operator(.MortLePlusJeune.)
    module procedure mort_le_plus_jeune
  end interface
  !-----------------------------------
  private :: operator(<), longevite, mort_le_plus_jeune, liste
contains
```

Corrigé de l'exercice 13 : solution avec liste chaînée double (suite)

```
subroutine init
  type(musicien)          :: mus
  type(musicien), pointer :: ptr_precedent, ptr_courant
  integer                 :: etat

  nullify(debut)
  nullify(mus%ptr_precedent)
  nullify(mus%ptr_suivant)
  open(1,file="musiciens",action="read",status="old")
  do
    read(1,"(a,1x,a,a,2(1x,i4))",iostat=etat) mus%prenom,      &
                                              mus%nom,         &
                                              mus%annee_naiss, &
                                              mus%annee_mort
    if (etat /= 0) exit
    allocate(ptr_courant)
    ptr_courant = mus
    if (.not.associated(debut)) then
      debut => ptr_courant
    else
      ptr_precedent%ptr_suivant => ptr_courant
      ptr_courant%ptr_precedent => ptr_precedent
    endif
    ptr_precedent => ptr_courant
  enddo
  close(1)
  print *
  print *,"---- Liste des musiciens ----"
  print *
  call liste
end subroutine init
```


Corrigé de l'exercice 13 : solution avec liste chaînée double (suite)

```
subroutine tri(critere)
  ! Procédure triant la liste des musiciens par ordre alphabétique des noms
  ! ou par ordre chronologique en fonction du paramètre « critere » spécifié.
  ! Déclaration de l'argument
  character(len=*), intent(in) :: critere
  ! Déclarations locales
  character(len=*), parameter, dimension(2) :: labels = &
          [ "alphabetique ", "chronologique" ]
  character(len=len (labels))               :: label_tri
  type(musicien), pointer                   :: ptr_courant, ptr
  logical                                   :: tri_termine, expr

  do
    tri_termine = .true.
    ptr_courant => debut
    do
      if(.not.associated(ptr_courant%ptr_suivant))exit
      select case(critere)
        case("nom")
          label_tri = labels(1)
          expr = ptr_courant%nom > ptr_courant%ptr_suivant%nom
        case("annee")
          label_tri = labels(2)
          expr = ptr_courant%annee_naiss > ptr_courant%ptr_suivant%annee_naiss
      end select
      if (expr) then
        allocate(ptr)
        ptr = ptr_courant%ptr_suivant
        call insere(ptr_courant, ptr)
        call suppression(ptr_courant%ptr_suivant)
        tri_termine = .false.
        cycle
      end if
      if (associated(ptr_courant%ptr_suivant)) ptr_courant => ptr_courant%ptr_suivant
    end do
    if (tri_termine) exit
  end do
  print *;print *,"---- Liste ",trim(label_tri)," des musiciens ----";print *;call liste
end subroutine tri
```

Corrigé de l'exercice 13 : solution avec liste chaînée double (suite)

```
subroutine insere(ptr_courant, ptr)
  type(musicien), pointer :: ptr_courant, ptr

  if (associated(ptr_courant, debut)) then
    debut => ptr
  else
    ptr_courant%ptr_precedent%ptr_suivant => ptr
  end if
  ptr%ptr_suivant => ptr_courant
  ptr%ptr_precedent => ptr_courant%ptr_precedent
  ptr_courant%ptr_precedent => ptr
end subroutine insere
!---------------------------
subroutine suppression(ptr)
  type(musicien), pointer :: ptr
  type(musicien), pointer :: temp

  temp => ptr ; ptr => ptr%ptr_suivant
  if (associated(temp%ptr_suivant)) &
                 temp%ptr_suivant%ptr_precedent => temp%ptr_precedent
  deallocate(temp)
end subroutine suppression
!---------------------------
function longevite(mus1,mus2)
  ! Fonction surchargeant l'opérateur < afin de pouvoir spécifier
  ! des opérandes de type « musicien ».
  type(musicien), intent(in) :: mus1,mus2
  logical                    :: longevite
  integer                    :: duree_de_vie_mus1, duree_de_vie_mus2

  duree_de_vie_mus1 = mus1%annee_mort - mus1%annee_naiss
  duree_de_vie_mus2 = mus2%annee_mort - mus2%annee_naiss
  longevite = duree_de_vie_mus1 < duree_de_vie_mus2
end function longevite
```

Corrigé de l'exercice 13 : solution avec liste chaînée double (suite)

```
function mort_le_plus_jeune(debut)
  type(musicien), intent(in) :: debut
  type(musicien)             :: mort_le_plus_jeune
  type(musicien), pointer    :: p_mus

  mort_le_plus_jeune = debut; p_mus => debut%ptr_suivant
  do while(associated(p_mus))
    ! Ici l'utilisation de l'opérateur < provoque l'appel à la fonction « longevite » :
    !    p_mus < mort_le_plus_jeune <=> longevite(p_mus, mort_le_plus_jeune)

    if (p_mus < mort_le_plus_jeune) mort_le_plus_jeune = p_mus
    p_mus => p_mus%ptr_suivant
  enddo
end function mort_le_plus_jeune
!----------------
function nom(mus)
  ! Retourne les nom et prénom du musicien passé en argument.
  type(musicien), intent(in) :: mus
  character(len=38)          :: nom

  nom = trim(mus%prenom)//" "//mus%nom
end function nom
```

Corrigé de l'exercice 13 : solution avec liste chaînée double (suite)

```
  subroutine liste
    type(musicien), pointer :: ptr_courant
    ptr_courant => debut
    if (.not.associated(debut)) then
      print *,"Il n'existe aucun musicien!"
      stop  8
    end if
    do
      write(*,"((5x,a,1x,a,2(1x,i4)))") ptr_courant%prenom,      &
                                        ptr_courant%nom,         &
                                        ptr_courant%annee_naiss, &
                                        ptr_courant%annee_mort
      if (.not.associated(ptr_courant%ptr_suivant)) exit
      ptr_courant => ptr_courant%ptr_suivant
    end do
    print *
    print *,"Liste inversée"
    print *,"--------------"
    print *
    do
      write(*,"((5x,a,1x,a,2(1x,i4)))") ptr_courant%prenom,      &
                                        ptr_courant%nom,         &
                                        ptr_courant%annee_naiss, &
                                        ptr_courant%annee_mort
      if (associated(ptr_courant, debut)) exit
      ptr_courant => ptr_courant%ptr_precedent
    end do
  end subroutine liste
end module music
```

Cette solution propose de trier la liste chaînée à l'aide d'un « **tableau de pointeur** ». Elle peut évidemment être adaptée pour s'appliquer à la liste chaînée simple. Notez que cette solution adopte le concept d'encapsulation.

Corrigé de l'exercice 13 : liste chaînée double (autre solution)

```
module music
  implicit none
  type musicien
    private
    character(len=16)       :: prenom
    character(len=21)       :: nom
    integer                 :: annee_naiss,annee_mort
    type(musicien), pointer :: ptr_precedent => NULL()
    type(musicien), pointer :: ptr_suivant   => NULL()
  end type musicien
  !
  type, private :: ptr_mus
    type(musicien), pointer :: ptr
  end type ptr_mus
  !-------------------------------------------------------------------
  type(musicien), pointer :: debut => NULL()
  type(ptr_mus), dimension(:), allocatable, private :: tab_ptr_musicien
  integer, private :: nb_mus
  !-------------------------------------------------------------------
  interface operator(<)
    module procedure longevite
  end interface
  !-----------------------------------
  interface operator(.MortLePlusJeune.)
    module procedure mort_le_plus_jeune
  end interface
  !-----------------------------------
  private :: operator(<), longevite, mort_le_plus_jeune, liste
```

Corrigé de l'exercice 13 : liste chaînée double (autre solution : suite)

```
contains
  subroutine init
    type(musicien)          :: mus
    type(musicien), pointer :: ptr_precedent, ptr_courant
    integer                 :: etat

    open(1,file="musiciens",action="read",status="old")
    nb_mus = 0
    do
      read(1,"(a,1x,a,2(1x,i4))",iostat=etat) mus%prenom,       mus%nom, &
                                              mus%annee_naiss, mus%annee_mort
      if (etat /= 0) exit
      allocate(ptr_courant); ptr_courant = mus
      if (.not.associated(debut)) then
        debut => ptr_courant
      else
        ptr_precedent%ptr_suivant => ptr_courant
        ptr_courant%ptr_precedent => ptr_precedent
      endif
      ptr_precedent => ptr_courant
      nb_mus = nb_mus + 1
    enddo
    close(1)
    call association
    print *; print *,"---- Liste des musiciens ----"; print *; call liste
  contains
    subroutine association
      integer i

      allocate( tab_ptr_musicien(nb_mus) )
      ptr_courant => debut
      do i=1,nb_mus
        tab_ptr_musicien(i)%ptr => ptr_courant
        ptr_courant => ptr_courant%ptr_suivant
      end do
    end subroutine association
  end subroutine init
```


Corrigé de l'exercice 13 : liste chaînée double (autre solution : suite)

```
  subroutine tri(critere)
    ! Procédure triant la liste des musiciens par ordre alphabétique des noms
    ! ou par ordre chronologique en fonction du paramètre « critere » spécifié.
    ! Déclaration de l'argument
    character(len=*), intent(in) :: critere
    ! Déclarations locales
    character(len=*), parameter, dimension(2) :: labels = &
            [ "alphabetique ", "chronologique" ]
    character(len=len(labels))                :: label_tri
    logical                                   :: expr
    integer                                   :: i, j

    do i=nb_mus-1,1,-1
      do j=1,i
        select case(critere)
          case("nom")
            label_tri = labels(1)
            expr = lgt(tab_ptr_musicien(j)%ptr%nom, tab_ptr_musicien(j+1)%ptr%nom)
          case("annee")
            label_tri = labels(2)
            expr = tab_ptr_musicien(j)%ptr%annee_naiss > &
                   tab_ptr_musicien(j+1)%ptr%annee_naiss
        end select
        if (expr) &
          !-------Permutation des deux associations-----------
          tab_ptr_musicien(j:j+1) = tab_ptr_musicien(j+1:j:-1)
          !---------------------------------------------------
      enddo
    enddo

    print *
    print *,"---- Liste ", trim(label_tri), " des musiciens ----"
    print *
    call liste
  end subroutine tri
```

Corrigé de l'exercice 13 : liste chaînée double (autre solution : suite)

```
  function longevite(mus1,mus2)
  ! Fonction permettant de surcharger l'opérateur < afin de pouvoir spécifier
  ! des opérandes de type « musicien ».
    type(musicien), intent(in) :: mus1,mus2
    logical                    :: longevite
    integer                    :: duree_de_vie_mus1,duree_de_vie_mus2

    duree_de_vie_mus1 = mus1%annee_mort - mus1%annee_naiss
    duree_de_vie_mus2 = mus2%annee_mort - mus2%annee_naiss
    longevite = duree_de_vie_mus1 < duree_de_vie_mus2
  end function longevite
  !--------------------------------
  function mort_le_plus_jeune(debut)
    type(musicien), intent(in) :: debut
    type(musicien)             :: mort_le_plus_jeune
    type(musicien), pointer    :: p_mus

    mort_le_plus_jeune = debut; p_mus => debut%ptr_suivant
    do while(associated(p_mus))
      ! Ici l'utilisation de l'opérateur < provoque l'appel à la fonction « longevite » :
      !    p_mus < mort_le_plus_jeune <=> longevite(p_mus, mort_le_plus_jeune)

      if (p_mus < mort_le_plus_jeune) mort_le_plus_jeune = p_mus
      p_mus => p_mus%ptr_suivant
    enddo
  end function mort_le_plus_jeune
```

Corrigé de l'exercice 13 : liste chaînée double (autre solution : suite)

```
  function nom(mus)
  ! Fonction renvoyant les nom et prénom du musicien passé en argument.
    type(musicien), intent(in) :: mus
    character(len=38)          :: nom

    nom = trim(mus%prenom)//" "//mus%nom
  end function nom
  !---------------
  subroutine liste
    integer i

    if ( nb_mus == 0 ) then
      print *,"Il n'existe aucun musicien!"
      stop  8
    end if
    do i=1,nb_mus
      write(*,"((5x,a,1x,a,2(1x,i4)))") tab_ptr_musicien(i)%ptr%prenom,       &
                                        tab_ptr_musicien(i)%ptr%nom,          &
                                        tab_ptr_musicien(i)%ptr%annee_naiss,  &
                                        tab_ptr_musicien(i)%ptr%annee_mort
    end do
    print *
    print *,"liste inversée"
    print *,"--------------"
    print *
    do i=nb_mus,1,-1
      write(*,"((5x,a,1x,a,2(1x,i4)))") tab_ptr_musicien(i)%ptr%prenom,       &
                                        tab_ptr_musicien(i)%ptr%nom,          &
                                        tab_ptr_musicien(i)%ptr%annee_naiss,  &
                                        tab_ptr_musicien(i)%ptr%annee_mort
    end do
  end subroutine liste
end module music
```

Procédures « pure »

Afin de faciliter l'optimisation et la parallélisation des codes, la norme 95 a prévu un nouvel attribut `pure` attaché aux procédures pour lesquelles ont peut garantir l'absence d'effet de bord (*side effect*). Elles pourront ainsi figurer au sein du « bloc `FORALL` » vu ci-après.

Le préfixe « `pure` » doit être ajouté à l'instruction `function` ou `subroutine`.

Exemple

```
pure function ftc(a,b)
  implicit none
  integer,intent(in) :: a, b
  real :: ftc
  ftc = sin(0.2+real(a)/(real(b)+0.1))
end function ftc
```

Voici brièvement, ce qui leur est interdit :

- modifier des entités (arguments, variables) vues de l'extérieur ;
- déclarer des variables locales avec l'attribut `SAVE` (ou ce qui revient au même les initialiser à la déclaration) ;
- faire des entrées/sorties dans un fichier externe.

Voici quelques règles à respecter :

- ne faire référence qu'à des procédures ayant aussi l'attribut `pure` et obligatoirement en mode d'interface explicite ;
- toujours définir la vocation (`intent`) des arguments muets (sauf ceux de type procédural bien sûr) : pour les fonctions cette vocation est obligatoirement `intent(in)` ;
- pour toute variable « vue » par *host* ou *use association* ou via `COMMON` ou via un argument muet avec `intent(in)` :
  - ne pas la faire figurer à gauche d'une affectation,
  - ne pas la faire figurer à droite d'une affectation si elle est de type dérivé contenant un pointeur,
  - ne pas la transmettre à une autre procédure si l'argument muet correspondant a l'un des attributs : `pointer`, `intent(out)`, `intent(inout)` ;
  - ne pas lui associer de pointeur.
- ne pas utiliser d'instruction `STOP` ;
- les fonctions (ou sous-programmes) surchargeant des opérateurs (ou l'affectation) doivent avoir l'attribut `pure`.

Remarques

- les fonctions intrinsèques ont toutes l'attribut `pure`,
- l'attribut `pure` est automatiquement donné aux procédures ayant l'attribut `elemental` (cf. ci-après).

Procédures « elemental »

Les procédures « `ELEMENTAL` » sont définies avec des arguments muets scalaires mais peuvent recevoir des arguments d'appels qui sont des tableaux du même type.

La généralisation du traitement scalaire à l'ensemble des éléments du/des tableaux passés ou retournés suppose bien sûr le respect des règles de conformance au niveau des profils (*shape*).

Voici les règles à respecter :

- nécessité d'ajouter le préfixe `ELEMENTAL` à l'instruction `function` ou `subroutine` ;
- l'attribut `ELEMENTAL` implique l'attribut `pure` ; il faut donc respecter toutes les régles énoncées au paragraphe précédent sur les procédures « `pure` » ;
- tous les arguments muets et la valeur retournée par une fonction doivent être des scalaires sans l'attribut `pointer` ;
- si un tableau est passé à un sous-programme « `ELEMENTAL` », tous les autres arguments à vocation `out`/`inout` doivent eux aussi être passés sous forme de tableaux et être conformants ;
- pour des raisons d'optimisation, un argument muet ne peut figurer dans une *specification-expr.* c.-à-d. être utilisé dans les déclarations pour définir l'attribut `DIMENSION` d'un tableau ou la longueur (`len`) d'une variable de type `character` ;
- l'attribut `ELEMENTAL` est incompatible avec l'attribut `RECURSIVE`.

Exemple

```
module mod1
 integer,parameter :: prec=selected_real_kind(6,30)
end module mod1

program P1
  USE mod1
  implicit none
  real(kind=prec)                    :: scal1,scal2
  real(kind=prec),dimension(1024)  :: TAB1 ,TAB2
       . . . . . .
  call permut(scal1,scal2)
       . . . . . .
  call permut(TAB1,TAB2)
       . . . . . .
contains
  elemental subroutine permut(x,y)
     real(kind=prec),intent(inout) :: x, y
     real                          :: temp
     temp = x
     x = y
     y = temp
  end subroutine permut
end program P1
```

Note : un opérateur surchargé ou défini via une fonction ELEMENTAL est lui même "élémentaire" ; il peut s'appliquer à des opérandes qui sont des tableaux de même type que ses opérandes scalaires.

Le « bloc FORALL »

```
[etiquette:]FORALL(index=inf:sup[:pas][,index=inf:sup[:pas]]...[,expr_logique_scalaire])
                Corps : bloc d'instructions
            END FORALL [etiquette]
```

Le « bloc FORALL » est utilisé pour contrôler l'exécution d'instructions d'affectation ou d'association (pointeur) en sélectionnant des éléments de tableaux via des triplets d'indices et un masque optionnel. Le bloc peut se réduire à une « instruction FORALL » s'il ne contient qu'une seule instruction.

Ce bloc a été défini pour faciliter la distribution et l'exécution des instructions du bloc, en parallèle sur plusieurs processeurs.

Sous le contrôle du « masque », chacune des instructions est interprétée de façon analogue à une "instruction tableau" ; les opérations élémentaires sous-jacentes doivent pouvoir s'exécuter simultanément ou dans n'importe quel ordre, l'affectation finale n'étant faite que lorsqu'elles sont **toutes** terminées.

La séquence des instructions dans le bloc est respectée.

La portée (*scope*) d'un indice (*index*) contrôlant un « bloc FORALL » est limitée à ce bloc. En sortie du bloc, une variable externe de même nom retrouve la valeur qu'elle avait avant l'entrée.

Exemples

1. Traitement particulier des lignes paires et impaires d'une matrice A(NL,NC) (NL pair) en excluant les éléments nuls. Le traitement des lignes impaires précède celui des lignes paires :

```
FORALL(i=1:NL-1:2, j=1:NC-1, A(i,j) /= 0.)
  A(i,j)   = 1./A(i,j)
  A(i+1,j)= A(i+1,j)*A(i,j+1)
END FORALL
```

   Avec une double boucle DO, les opérations élémentaires et les affectations se feraient dans l'ordre strict des itérations : les résultats seraient différents.

2. Inversion de chaque ligne du triangle inférieur d'une matrice carrée d'ordre N :

```
exter:FORALL(i=2:N)
        inter:FORALL(j=1:i)
                A(i,j) = A(i,i-j+1)
              END FORALL inter
      END FORALL exter
```

3. Forme plus condensée en considérant chaque ligne comme une section régulière et en adoptant la syntaxe de l'« instruction FORALL » :

```
FORALL(i=2:N) A(i,1:i) = A(i,i:1:-1)
```


Transformation ligne par ligne d'un tableau A de N lignes et stockage du résultat dans le tableau B. Utilisation d'un bloc WHERE et appel de fonctions intrinsèques ou ayant l'attribut « pure » dans le corps du bloc FORALL.

Exemples (suite)

```
program transform
  implicit none
  integer,parameter     :: N=5, M=8
  real, dimension(N,M)  :: A, B
        . . . .
  FORALL(i=1:N)
    WHERE(abs(A(i,:)) <= epsilon(+1.)) &
        A(i,:) = sign(epsilon(+1.),A(i,:))
    B(i,:) = ftc(i,N) / A(i,:)
  END FORALL
        . . . .
 contains
  pure function ftc(a,b)
    integer,intent(in) :: a, b
    real               :: ftc
    ftc = sin(0.2+real(a)/(real(b)+0.1))
  end function ftc
end program transform
```

Aspects obsolètes : fortran 90

1. IF arithmétique [2] : `IF (ITEST) 10,11,12`
   ==> `IF--THEN--ELSE IF--ELSE--ENDIF`
2. branchement au `END IF` depuis l'extérieur [1]
   ==> se brancher à l'instruction suivante.
3. boucles `DO` pilotées par un réel [2] : `DO 10 R=1., 5.7, 1.3`
4. partage d'une instruction de fin de boucle [1] :

```
      DO 1 I=1,N
        DO 1 J=1,N
          A(I,J)=A(I,J)+C(J,I)
    1 CONTINUE
```

   ==> autant de `CONTINUE` que de boucles.
5. fins de boucles autres que `CONTINUE` ou `END DO`

1. déclaré hors norme par Fortran 95
2. déclaré hors norme par Fortran 2018

Aspects obsolètes : fortran 90

1. ASSIGN [1] et le GO TO assigné [1] :

```
ASSIGN 10 TO intvar
....
ASSIGN 20 TO intvar
....
GO TO intvar
```

   ==> SELECT CASE ou IF/THEN/ELSE

2. ASSIGN [1] d'une étiquette de FORMAT :

```
  ASSIGN 2 TO NF           CHARACTER(7),DIMENSION(4)::C
2 FORMAT (F9.2)     ==>    I = 2; C(2) = '(F9.2)'
  PRINT NF,TRUC            PRINT C(I),TRUC
```

1. déclaré hors norme par Fortran 95

Aspects obsolètes : fortran 90

1. RETURN multiples :

```
      CALL SP1(X,Y,*10,*20)
          ...
10        ...
          ...
20        ...
          ...
      SUBROUTINE SP1(X1,Y1,*,*)
          ...
      RETURN 1
          ...
      RETURN 2
          ...
```

   ==> SELECT CASE sur la valeur d'un argument retourné

2. PAUSE [1] 'Montez la bande 102423 SVP'
   ==> READ qui attend les données

3. FORMAT(9H A éviter) [1]
   ==> Constante littérale : FORMAT(' Recommandé')

1. déclaré hors norme par Fortran 95

Aspects obsolètes : fortran 95

1. le « format fixe » du source
   ==> « format fixe » ;
2. le GO TO calculé (GO TO (10,11,12,...), *int_expr*)
   ==> select case ;
3. l'instruction DATA placée **au sein** des instructions exécutables
   ==> **avant** les instructions exécutables ;
4. « *Statement functions* » (sin_deg(x)=sin(x*3.14/180.))
   ==> procédures internes ;
5. le type CHARACTER*... dans les déclarations
   ==> CHARACTER(LEN=...) ;
6. le type CHARACTER(LEN=*) de longueur implicite en retour d'une fonction
   ==> CHARACTER(LEN=len(str)).

Aspects obsolètes : fortran 2008/2018

1. instruction ENTRY [1].
2. structure de contrôle FORALL [2] ;
3. instruction EQUIVALENCE [2] ;
4. bloc COMMON [2] ;
5. unité de programme BLOCK DATA [2] ;
6. boucle labellée [2] :

```
      DO 10 I=1,N
         ...
10 CONTINUE
```

7. appel aux procédures intrinsèques autrement que par leur nom générique [2] :
   - iabs, abs, dabs, cabs ==> abs
   - max0, amax1, dmax1 ==> max
   - real, float, sngl ==> real
   - ...

1. déclaré obsolète par Fortran 2008
2. déclaré obsolète par Fortran 2018

– Symboles –

– A –

– B –

– C –

– D –

– E –

– F –

– G –

– H –

– I –

– K –

– L –

– M –

– N –

– O –

– P –

– R –

– S –

– T –

– U –

– V –

– W –

– Z –