Cours Fortran 95

Patrick CORDE et Hervé DELOUIS

23 mai 2006

Version 9.4

Cours Fortran 95

Constitué à partir des *transparents* du cours Fortran 95 de l'IDRIS, ce manuel ne traite que des nouveautés importantes apportées par les normes 90 et 95. Il suppose donc une bonne connaissance préalable de Fortran 77. À noter que l'IDRIS propose aussi un cours Fortran 95 1erniveau ainsi qu'un support de cours Fortran 2003 (en préparation). Pour une étude exhaustive, consulter les manuels référencés aux paragraphes 1.7 et 1.8.

Patrick CORDE **<corde@idris.fr>**

Hervé DELOUIS **<delouis@idris.fr>**

29 mai 2006

I001 267p

Table des matières

1 Introduction

⇒ Historique

⇒ Compatibilité norme 77/90

⇒ Apports de Fortran 90

⇒ Aspects obsolètes de Fortran 90

⇒ Aspects obsolètes de Fortran 95

⇒ Évolution : nouveautés Fortran 95

⇒ Bibliographie

⇒ Documentation

1.1 Historique

- Code machine (notation numérique en octal).
- Assembleurs de codes mnémoniques.
- 1954–Projet création du premier langage symbolique par John Backus d'IBM $\Longrightarrow$ **FORTRAN** (*Mathematical* **FOR***mula* **TRAN***slating System*) :
  1. Efficacité du code généré (performance).
  2. Langage quasi naturel pour scientifiques (productivité, maintenance, lisibilité).
- 1957–Livraison des premiers compilateurs.
- 1958–**Fortran II** (IBM) $\Longrightarrow$ sous-programmes compilables de façon indépendante.
- Généralisation aux autres constructeurs mais :
  – divergences des extensions $\Longrightarrow$ nécessité de **normalisation**,
  – ASA *American Standards Association* ($\Longrightarrow$ ANSI *American Nat. Standards Institute*). Comité chargé du développement d'une norme Fortran $\Longrightarrow$ 1966.

Introduction : historique

- 1966–**Fortran 66** (Fortran IV). Première norme.
- Évolution par extensions divergentes. . .
- 1977–**Fortran 77** (Fortran V). Quasi compatible :
  aucune itération des boucles *nulles* $\Longrightarrow$ `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 ISO et ANSI $\Longrightarrow$ **Fortran 90**
- 1994 – Premiers compilateurs Fortran 90 Cray et IBM.
- 1997 – Norme ISO et ANSI $\Longrightarrow$ **Fortran 95**
- 1999 – sur Cray T3E puis IBM RS/6000 $\Longrightarrow$ **Fortran 95**
- septembre 2004 – Norme ISO et ANSI $\Longrightarrow$ **Fortran 2003**

1.2 Compatibilité norme 77/90

- La norme 77 est totalement incluse dans la norme 90.
- Quelques comportements différents :
  - beaucoup plus de fonctions/sous-progr. intrinsèques $\Longrightarrow$ risque d'homonymie avec procédures externes Fortran 77 et donc de résultats différents ! $\Longrightarrow$ **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 !

1.3 Apports de Fortran 90

- Procédures internes (**CONTAINS**), **modules** (**USE**).
- “Format libre”, identificateurs, déclarations, `!`, `&`, `;`.
- Précision des nombres : **KIND** $\Longrightarrow$ 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**.

1.4 Aspects obsolètes de Fortran 90

Notations

((H.N.95)) : aspects devenus Hors Norme 95

$\Longrightarrow$: *conseil de substitution*

1. IF arithmétique : **`IF (ITEST) 10,11,12`**

   $\Longrightarrow$ **`IF--THEN--ELSE IF--ELSE--ENDIF`**

2. Branchement au **`END IF`** depuis l'extérieur *((H.N.95))*

   $\Longrightarrow$ se brancher à l'instruction suivante.

3. Boucle **`DO`** pilotée par réel : **`DO 10 I=1., 5.7, 1.3`** *((H.N.95))*

4. Partage d'une instruction de fin de boucle :

```
      DO 1 I=1,N
        DO 1 J=1,N
          A(I,J)=A(I,J)+C(J,I)
    1 CONTINUE
```

   $\Longrightarrow$ autant de **`CONTINUE`** que de boucles.

5. Fins de boucles autres que **`CONTINUE`** ou **`END DO`**

6. **`ASSIGN`** et le **`GO TO`** assigné : *((H.N.95))*

```
ASSIGN 10 TO intvar
....
ASSIGN 20 TO intvar
....
GO TO intvar
```

   $\Longrightarrow$ **`SELECT CASE`** ou **`IF/THEN/ELSE`**

7. **ASSIGN** d'une étiquette de **FORMAT** : *((H.N.95))*

```
ASSIGN 2 TO NF                    CHARACTER(7),DIMENSION(4)::C
2 FORMAT (F9.2)          ⟹       I=2
PRINT NF,TRUC                     C(2)='(F9.2)'
                                  PRINT C(I),TRUC
```

8. **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

9. **PAUSE 'Montez la bande 102423 SVP'** *((H.N.95))*

⟹ **READ** qui attend les données

10. **FORMAT(9H A éviter)** *((H.N.95))*

⟹ Constante littérale : **FORMAT(' Recommandé')**

1.5 Aspects obsolètes de Fortran 95

1. Le “format fixe” du source
   $\Longrightarrow$ “format libre”

2. Le **`GO TO`** calculé (**`GO TO (10,11,12,...),`** *int_expr*)
   $\Longrightarrow$ **`select case`**

3. L’instruction **`DATA`** placée **au sein** des instructions exécutables
   $\Longrightarrow$ **avant** les instructions exécutables

4. *Statement functions* (**`sin_deg(x)=sin(x*3.14/180.)`**)
   $\Longrightarrow$ procédures internes

5. Le type **`CHARACTER*...`** dans les déclarations
   $\Longrightarrow$ **`CHARACTER(LEN=...)`**

6. Le type **`CHARACTER(LEN=*)`** de longueur implicite en retour d’une fonction
   $\Longrightarrow$ **`CHARACTER(LEN=len(str))`**

1.6 Évolution : principales nouveautés Fortran 95

Le processus de normalisation (comité X3H5 de l'ANSI) se poursuit ; notamment des extensions seront proposées pour machines parallèles à mémoire distribuée (HPFF : *Hight Perf. Fortran Forum*) ou partagée (OpenMP-2).

- **`FORALL(i=1:n,j=1:m,y(i,j)/=0.) x(i,j)=1./y(i,j)`** (cf. Annexe C3 page 262).
- 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 C1 page 258 et Annexe C2 page 260).
- Fonction intrinsèque **`NULL()`** pour forcer un pointeur à l'état nul y compris lors de sa déclaration (cf. chap. 7.1 page 118).
- Libération automatique des tableaux dynamiques locaux n'ayant pas l'attribut **`SAVE`** (cf. chap. 6.2 page 113).
- Valeur initiale par défaut pour les composantes d'un type dérivé (cf. chap. 3.1 page 42).
- Fonction intrinsèque **`CPU_TIME`** (cf. chap. 14.6 page 211).
- Bloc **`WHERE`** : imbrication possible (cf. chap. 5.6 page 104).
- Expressions d'initialisation étendues (cf. chap. 5.7 page 107).
- **`MAXLOC`**/**`MINLOC`** : ajout argument **`dim`** (cf. Chap. 5.5.1 page 79).
- Ajout **`generic_spec`** (cf. Chap. 9 page 155)
  dans **`END INTERFACE [generic_spec]`**
- Suppressions (cf. page 14)... et nouveaux aspects obsolètes.

1.7 Bibliographie

- ADAMS, BRAINERD, MARTIN, SMITH et WAGENER, *Fortran 95 Handbook*, MIT PRESS, 1997, (711 pages), ISBN 0-262-51096-0.
- BRAINERD, GOLDBERG, ADAMS, *Programmer's guide to Fortran 90*, 3^{e}édit. UNICOMP, 1996, (408 pages), ISBN 0-07-000248-7.
- 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.
- HAHN B.D., *Fortran 90 for the Scientist & Engineers*, Edward Arnold, London, 1994, (360 pages), ISBN 0-340-60034-9.
- 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, (320pages), ISBN 2-225-85229-4.
- LIGNELET P., *Structures de données et leurs algorithmes avec Fortran 90 et Fortran 95*, MASSON, 1996, (360pages), ISBN 2-225-85373-8.

Introduction : bibliographie

- METCALF M., REID J.,
  - *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 90/95 explained*, Oxford University Press, 1996, (345 pages), ISBN 0-19-851888-9.
- METCALF M., REID J. and COHEN M., *"Fortran 95/2003 Explained"*, fin 2004, Oxford University Press, (ISBN 0-19-852693-8/0-19-852692-X).
- MORGAN and SCHOENFELDER, *Programming in Fortran 90*, Alfred Waller Ltd., 1993, ISBN 1-872474-06-3.
- 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.

1.8 Documentation

- **IBM/SP4** $\Longrightarrow$ **http://www.idris.fr/su/Scalaire/zahir/index.html**
  - descriptif matériel et logiciel,
  - système de compilation Fortran,
  - exécution d'un code en intéractif et en batch,
  - débogage,
  - analyse de performances,
  - documentation Fortran **`IBM`**,
  - FAQ.
- **NEC/SX5** $\Longrightarrow$ **http://www.idris.fr/su/Vectoriel/uqbar/index.html**
  - descriptif matériel et logiciel,
  - système de compilation Fortran,
  - exécution d'un code en intéractif et en batch,
  - débogage,
  - analyse de performances,
  - documentation Fortran **`NEC`** (accès limité aux utilisateurs **`IDRIS`**),
  - FAQ.
- **Documentation générale**
  - Support de cours (en préparation) Fortran 2003,
  - Supports des cours Fortran 95 IDRIS (niveau 1 et 2),
  - Manuel "Fortran 77 pour débutants" (en anglais) $\Longrightarrow$ **http://www.idris.fr/data/cours/lang/fortran/choix_doc.html**
  - *The Fortran Company* $\Longrightarrow$ **http://www.swcp.com/~walt/**

2 Généralités

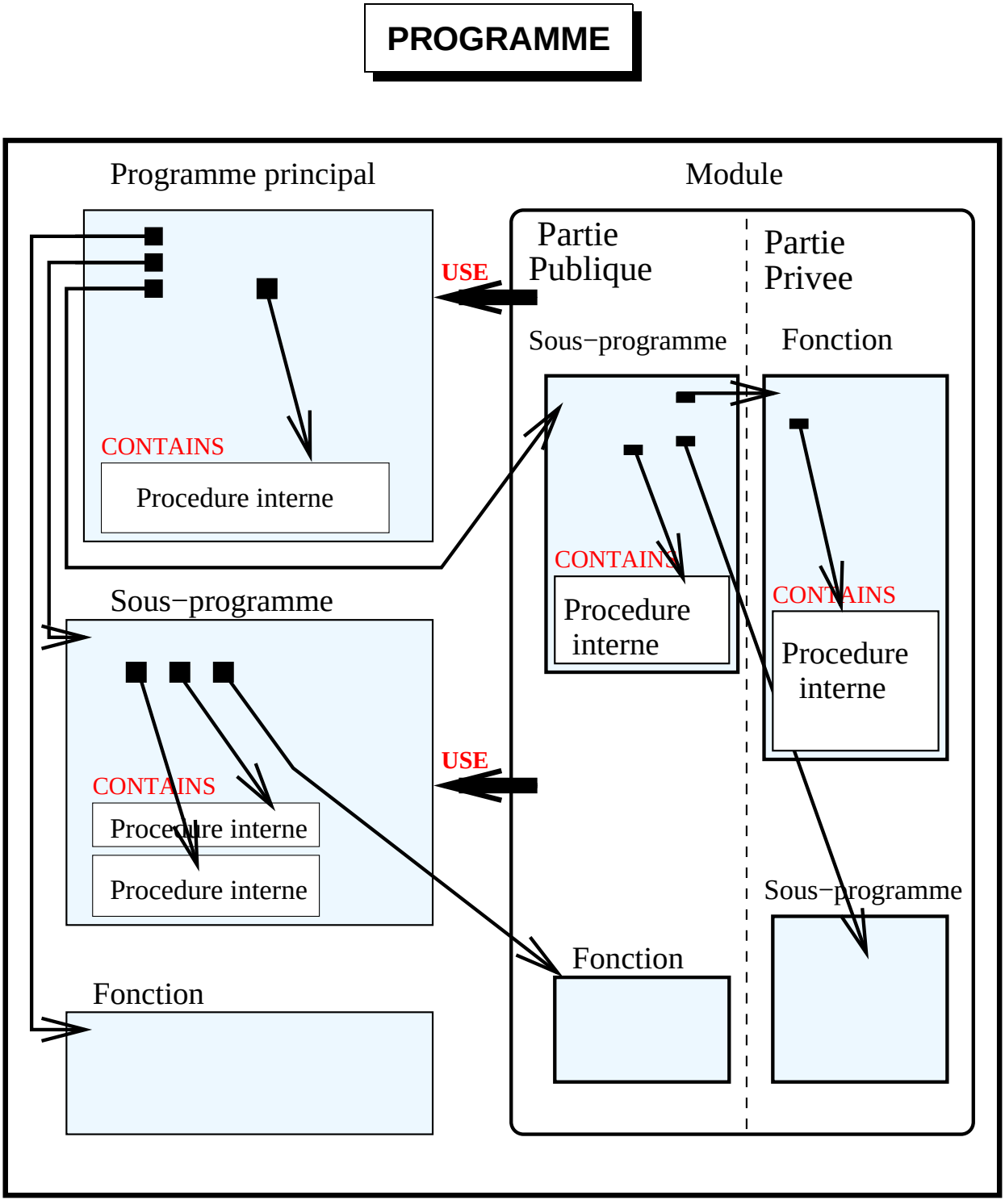
⇒ Structure d'un programme

⇒ Éléments syntaxiques

- Les identificateurs
- Le "format libre"
- Les commentaires
- Le "format fixe"
- Les déclarations
- Typage et précision des nombres : paramètre KIND

⇒ Compilation, édition des liens, exécution

2.1 Structure d'un programme



2.2 Éléments syntaxiques

2.2.1 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 **31** caractères.
On ne distingue pas les **majuscules** des **minuscules**.

Attention : en "format libre", les blancs sont significatifs.

Exemples d'identificateurs :

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

En **Fortran 95** il existe un certain nombre de **mots-clés** (`real`, `integer`, `if`, `logical`, `do`, ...), 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.

2.2.2 Le “format libre”

Dans le mode “**format libre**” les lignes peuvent être de longueur quelconque à concurrence de **132** caractères.
Il est également possible de coder plusieurs instructions sur une même ligne en les séparant avec le caractère **;**.

Exemple :

```
print *, ’ Entrez une valeur :’; read *,n
```

Une instruction peut être codée sur plusieurs lignes : on utilisera alors le caractère **&**.

Exemple :

```
print *, ’Montant  HT :’, montant_ht,  &
         ’        TVA :’, tva       ,  &
         ’Montant TTC :’, montant_ttc
```

Lors de la coupure d’une chaîne de caractères la suite de la chaîne doit obligatoirement être précédée du caractère **&**.

Exemple :

```
print *, ’Entrez un nombre entier &
        &compris entre 100 & 199’
```

2.2.3 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) ! 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.

Généralités : syntaxe
Les commentaires

Par contre, il n'est pas possible d'insérer un commentaire entre deux instructions situées sur une même ligne. Dans ce cas la 2^{e}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 90 en "**format libre**" et génèrent des erreurs de compilation.

2.2.4 Le “format fixe”

Le “**format fixe**” de **Fortran 95** 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.

2.2.5 Les déclarations

Forme générale d'une déclaration

type[, liste_attributs : :] liste_objets

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

Exemples 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
```

Voici deux déclarations équivalentes d'un pointeur **ptr** susceptible d'être associé à un tableau de réels :

```
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 sur plusieurs instructions. Par exemple :

```
integer tab
dimension tab(10)
target tab
```

Dans ce dernier exemple, il est plutôt recommandé de regrouper l'ensemble des attributs sous la forme :

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

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

Exemples :

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

Attention : 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**.

Par contre, en **Fortran 90** toute variable initialisée est permanente ; elle reçoit l'attribut **`save`** implicitement.

Typage par défaut : mêmes règles qu'en **Fortran 77**

- il est vivement recommandé d'utiliser l'instruction **`IMPLICIT NONE`**
- types prédéfinis (ou intrinsèques) : **`REAL, INTEGER, COMPLEX, LOGICAL, CHARACTER`**
- types-dérivés définis par le développeur.

2.2.6 Typage et précision des nombres : paramètre KIND

Réel	simple précision	4 ou 8 octets
Réel	double précision	8 ou 16 octets
Réel	quadruple précision	32 octets ?

Les types prédéfinis en **Fortran 90** sont en fait 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. Cette valeur désigne la **variante** souhaitée pour un **type** donné.

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.

En ce qui concerne 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.

Exemples :

- **`real(kind=8) x`**

  Réel double précision sur IBM RS/SP4 ou NEC SX5 et simple précision sur Cray T3E.

  C'est l'équivalent du **`real*8`** souvent utilisé en Fortran 77.

- **`integer(kind=2), target, save :: i`**

  équivalent de l'extension **`integer*2`** en Fortran 77.

À chaque type correspond une **variante** par défaut, sélectionnée en l'absence du paramètre **`KIND`** : c'est par exemple, la simple précision pour les réels.

($\Longrightarrow$ Voir tableau des sous-types sur IBM RS/SP4 et NEC SX5 en annexe A page 218)

Généralités : syntaxe
La précision des nombres (kind)

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.

Exemples de constantes numériques :

```
23564_4
12.879765433245_8
```

ou ce qui est plus portable :

```
integer, parameter :: short = 2, long = 8
1234_short
12.879765433245_long
```

Exemples de constantes chaînes de caractères :

```
1_'wolfy'
2_"wolfy"
integer(kind=short), parameter :: kanji = 2
kanji_"wolfy"
```

Fonction intrinsèque KIND

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

Exemples :

`kind(1.0)`	retourne la valeur associée au **sous-type** réel simple précision.
`kind(1.0d0)`	retourne la valeur associée au **sous-type** réel double précision.
`real(kind=kind(1.d0))`	permet de déclarer un réel double précision quelle que soit la machine utilisée.
`integer(kind=kind(0))`	pour déclarer un entier correspondant au **sous-type** entier par défaut.
`character(kind=kind('A'))`	pour déclarer une variable caractère avec le **sous-type** `character` par défaut.

***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).

Généralités : syntaxe
La précision des nombres (kind)

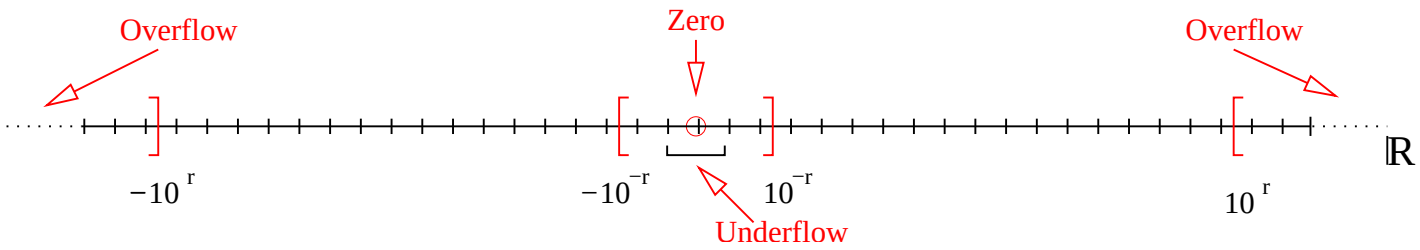
Fonction intrinsèque SELECTED_INT_KIND(r)

Elle 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.

Fonction intrinsèque SELECTED_REAL_KIND(p,r)



Elle 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 :

$$10^{-r} < |x| < 10^r$$

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.

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
  . . . .
```

Autres exemples d'appel :

```
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.

Généralités : syntaxe
La précision des nombres (kind)

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 :

$$|\ entier\ | < 10^{r}$$

$$10^{-r} < |\ réel\ | < 10^{r}$$

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

2.3 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/SP4, 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.f`** 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`**

Page réservée pour vos notes personnelles...

3 Types dérivés

⇒ Définition et déclaration de structures

⇒ Initialisation (constructeur de structure)

⇒ Symbole % d'accès à un champ

⇒ Types dérivés et procédures

⇒ Types dérivés et entrées/sorties

⇒ Conclusion et rappels

Types dérivés : définition

3.1 Définition et déclaration de structures

- **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=16) :: nom
  real,dimension(3) :: compos
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', (/ 1.,0.,0. /)),    &
   couleur('vert ', (/ 0.,1.,0. /)),    &
   couleur('bleu ', (/ 0.,0.,1. /))  /)
```

3.2 Initialisation (constructeur de structure)

Dans l'expression

```
(/ couleur('rouge', (/ 1.,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 **/)** permettant l'initialisation ou l'affectation globale d'un tableau de rang 1.

Exemples :

```
real,dimension(3)          :: TJ
type(couleur),dimension(5) :: TC
TC(1)=couleur('gris_fonce', (/ 0.2,0.2,0.2 /))
TC(2)=couleur('gris_clair', (/ 0.8,0.8,0.8 /))
TJ=(/ 1.0, 1.0, 0.0 /)
TC(3)=couleur('jaune',TJ)
```

3.3 Symbole % d'accès à un champ

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

- **`TC`** $\Longrightarrow$ tableau de structures de données de type dérivé **`COULEUR`**.
- **`TC(2) et TABRVB(3)`** $\Longrightarrow$ structures de type **`COULEUR`**.
- **`TC(1)%nom`** $\Longrightarrow$ champ **`nom`** (**`"gris_fonce"`**) de **`TC(1)`**.
- **`TC(1)%compos`** $\Longrightarrow$ tableau de 3 réels contenant les composantes RVB de la teinte **`gris_fonce`**.
- **`TC(2)%compos(2)`** $\Longrightarrow$ réel : composante verte du **`gris_clair`**.
- **`TC%compos(2)`** $\Longrightarrow$ tableau de 5 réels : composantes vertes. **Attention** : dans le cas où l'opérande de gauche est un tableau (ici **`TC`**), l'opérande de droite ne doit pas avoir l'attribut **`pointer`** !
- **`TC%compos`** $\Longrightarrow$ INCORRECT ! ! car au moins une des deux entités encadrant le **%** doit être un scalaire (rang nul) sachant qu'une structure est considérée comme un scalaire. Dans ce cas, **`TC`** et **`compos`** sont des tableaux de rang 1.

Types dérivés : symbole %

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

```
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 % .

Voici donc une nouvelle définition de la couleur jaune :

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

ou plus simplement :

```
TC(4)=couleur('jaune',    &
 tabrvb(1)%compos    + tabrvb(2)%compos)
```

3.4 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** $\Longrightarrow$ 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.

```
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=16) :: nom
  real,dimension(3) :: compos
end type COLOR
!-----------------------------------------
type(COLOR) :: col_in, demi_teinte
demi_teinte%nom=trim(col_in%nom)//'_demi'
demi_teinte%compos=col_in%compos/2.
end function demi_teinte
```

```
program geom3d
  implicit none
  integer :: i
  type VECTEUR
    real        :: x,y,z
  end type VECTEUR
  type CHAMP_VECTEURS ! >>>> Types imbriqués
    integer     :: n  !      Nb. de vecteurs
    type(VECTEUR),dimension(20) :: vect !taille
  end type CHAMP_VECTEURS                !max.
!------------------------Déclarations --------
  type(VECTEUR)                :: u,v,w
  type(CHAMP_VECTEURS)         :: champ
  real                         :: ps
!--------------------------------------------
  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=1,18) /)!>>> Construct.
  .........                      !    tableau
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 procédures

Exemple de définition d'un type dérivé **OBJ_MAT** et d'une fonction **som_mat** réalisant la somme de deux structures de ce type :

```
program testmat
 implicit none
 type OBJ_MAT
  integer :: N, M  ! >>>> Matrice ( N x M )
  real,dimension(:,:),allocatable:: A !-Erreur !
 end type OBJ_MAT
 !-----------------------Déclarations ---------
 type(OBJ_MAT)                :: MAT1, MAT2, MAT3
 integer                      :: NL,NC
 !---------------------------------------------
 read *, NL, NC
 MAT1%N=NL; MAT1%M=NC; allocate(MAT1%A(NL,NC))
 MAT2%N=NL; MAT2%M=NC; allocate(MAT2%A(NL,NC))
 read *, MAT1%A, MAT2%A
 MAT3 = som_mat(MAT1,MAT2)
contains
 function som_mat(mat1,mat2)
  type(OBJ_MAT),intent(in) :: mat1,mat2
  type(OBJ_MAT)            :: som_mat
  if(mat1%M /= mat2%M .or. mat1%N /= mat2%N)then
   stop 'ERREUR: profils différents'
  else
   som_mat%N=mat1%N ; som_mat%M=mat1%M
   allocate(som_mat%A(mat1%N , mat1%M))
   som_mat%A = mat1%A + mat2%A
  end if
 end function som_mat
end program testmat
```

Attention : attribut **ALLOCATABLE** $\Longrightarrow$ solution via pointeur et allocation dynamique (cf. remarque en fin de chapitre).

3.5 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=16)  :: nom=" "
  real, dimension(3) :: compos= (/ 0., 0., 0. /)
end type couleur
!-----------------------------------------------
integer                          :: long
type(couleur), dimension(5) :: tc
. . . .
inquire(iolength=long) tc, long
print *, "tc=", tc, "long=", long
. . . .
!---Écriture avec format
open(unit=10, file='F10', form='formatted')
write(unit=10, fmt="(5(A16,3E10.3),I4)") tc, long
. . . .
!---Écriture sans format (binaire)
open(unit=11, file='F11', form='unformatted')
write(unit=11) tc, long
. . . .
```

3.6 Conclusion et rappels

Pour faciliter la manipulation des structures de données (objets de type dérivé) nous verrons qu'il est possible par exemple :

- de définir d'autres fonctions de manipulation de matrices (addition, soustraction, multiplication, inversion, etc.).
- d'*encapsuler* le type et les fonctions opérant dessus dans un **module** séparé pour pouvoir y accéder plus facilement et plus sûrement dans toutes les unités de programme en ayant besoin,
- de définir des opérateurs génériques plus naturels en surchargeant les opérateurs `+` , `-` , `*` ou même de nouveaux tels que `.TRANSP.` (pour une expression du type `B=X+(.TRANSP.A)`, en les associant aux fonctions correspondantes (`interface OPERATOR`),
- de redéfinir le symbole d'affectation `=` pour créer une matrice à partir d'un vecteur ou obtenir sa taille (`interface ASSIGNMENT`),
- de cacher (`PRIVATE`) les composantes internes d'une structure.

Sans être un vrai *langage orienté objet*, Fortran 95 fournit ainsi des *extensions objet* bien utiles pour le confort et la fiabilité. Les notions manquantes de *classe* (hiérarchie de types dérivés extensibles avec héritage) et de *polymorphisme dynamique* applicable aux objets et aux méthodes/opérateurs relatifs à une *classe* font partie des propositions de la future norme **Fortran 2003**.

Types dérivés : conclusion

Rappels :

- 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 n'est possible qu'en Fortran 95,
- l'attribut **`ALLOCATABLE`** est interdit au niveau d'un champ, mais un tableau de structures peut avoir l'attribut **`ALLOCATABLE`** pour être alloué dynamiquement,
- 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`**.

Types dérivés : conclusion

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 129 ;
- 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. 7.12 page 134 et le corrigé de l'exercice 8 en annexe B ;
- l'allocation dynamique de mémoire appliquée à un champ de structure (l'attribut **`allocatable`** y étant interdit) —
  cf. paragraphe *Allocation dynamique de mémoire* du chapitre 7 *Pointeurs* et l'exemple du chapitre 10 *Surcharge d'opérateurs*.

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.

4 Programmation structurée

⇒ Introduction

⇒ Boucles **DO**

⇒ Construction **SELECT-CASE**

4.1 Introduction

Structure habituelle d'un programme en blocs :

Exemple :

```
[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.

4.2 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
```

Programmation structurée
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
```

3^{e}forme :

Ce sont des boucles DO sans contrôle de boucle.

Pour en sortir $\Longrightarrow$ instruction conditionnelle avec instruction **EXIT** dans le corps de la boucle.

```
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 évidemment pas l'optimisation,
- suivant que le test de sortie est fait en début ou en fin, cette boucle s'apparente au **DO WHILE** ou au **DO UNTIL**.

Programmation structurée
Boucles DO

Bouclage anticipé $\Longrightarrow$ 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, l'**EXIT** et le **CYCLE** peuvent être étiquetés pour s'appliquer à la boucle portant l'étiquette (*do-construct-name*) spécifiée.

Instructions EXIT et CYCLE dans des boucles imbriquées

$\Longrightarrow$ Utilisation de boucles étiquetées.

Exemple 1 :

```
          implicit none
          integer                 :: i, l, 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
```

Programmation structurée
Boucles DO

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
```

4.3 Construction SELECT-CASE

"Aiguillage" : équivalent du **CASE** de PASCAL et du **SWITCH** de C.

Exemple :

```
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
```

Note : le bloc **SELECT CASE** ne peut s'appliquer qu'à une expression scalaire (*case-expr*) de type entier, logique ou chaîne de caractères.

Programmation structurée
Construction SELECT-CASE

Page réservée pour vos notes personnelles...

5 Extensions tableaux

- ⇒ Définitions (rang, profil, étendue,...)
- ⇒ Manipulations de tableaux (conformance, constructeur, section, taille,...)
- ⇒ Tableau en argument d'une procédure (taille et profil implicites)
- ⇒ Section de tableau non contiguë en argument d'une procédure
- ⇒ Fonctions intrinsèques tableaux
  - – Interrogation (**`maxloc, lbound, shape,..`**)
  - – Réduction (**`all, any, count, sum,...`**)
  - – Multiplication (**`matmul, dot_product,...`**)
  - – Construction/transformation (**`reshape, cshift, pack, spread, transpose,...`**)
- ⇒ Instruction et bloc **`WHERE`**
- ⇒ Expressions d'initialisation autorisées
- ⇒ Quelques exemples d'expressions tableaux

5.1 Définitions (rang, profil, étendue,...)

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 :

Exemples :

```
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 !

Exemples :

```
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**.

5.2 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**.

Exemples d'expressions de type tableau :

```
integer, dimension(4,3) :: a
a = 1
```

L'expression précédente permet d'affecter l'entier **1** à tous les éléments du tableau **a**. Cette affectation est possible car
un scalaire est supposé **conformant** à tout tableau.

```
real,    dimension(6,7)      :: a,b
real,    dimension(2:7,5:11) :: c
logical, dimension(-2:3,0:6) :: l
b = 1.5
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 des exemples précédents :

```
real,    dimension(6,7)      :: a,b
real,    dimension(2:7,5:11) :: c
logical, dimension(-2:3,0:6) :: l
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 ***/)***.

Exemples :

```
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. Exemple :

```
(((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.

Sections régulières

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.

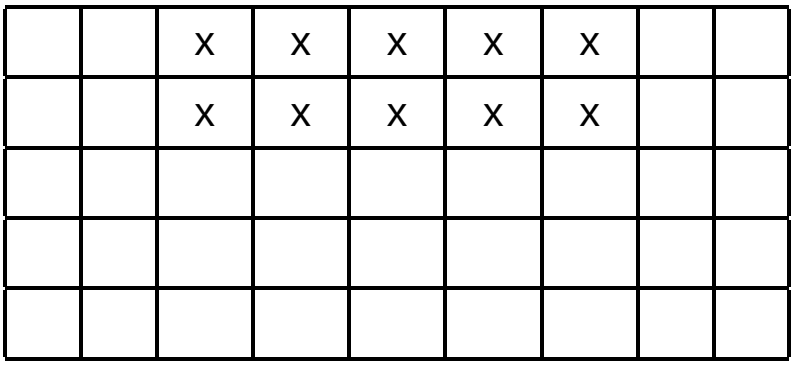
Par défaut, la valeur du **`pas`** est 1 et les valeurs de **`val_init`** et **`val_fin`** sont les limites définies au niveau de la déclaration du tableau parent. La notation *dégénérée* sous la forme d'un simple " :" correspond à l'étendue de la dimension considérée.

Exemples :

```
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"
```

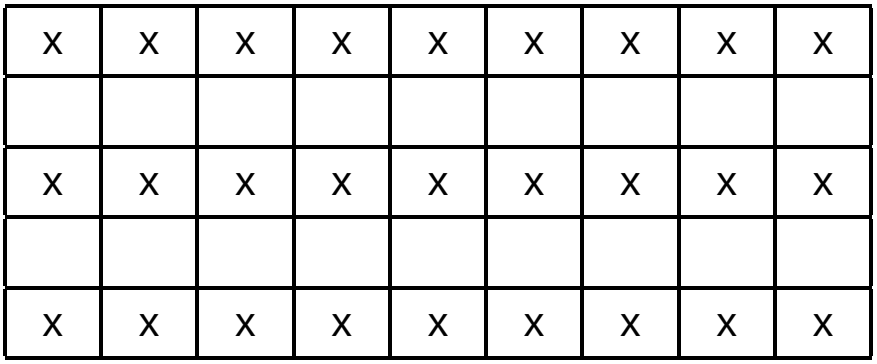
Extensions tableaux
Manipulations de tableaux

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



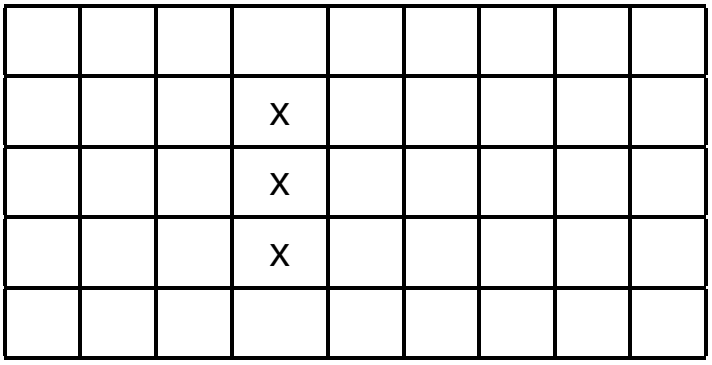
← **T(1:2,3:7)**

Section régulière de rang 2 et de profil (/ 2,5 /).



← **T(1:5:2,:)**

Section régulière de rang 2 et de profil (/ 3,9 /).



← **T(2:4,4)**

Section régulière de rang 1 (un indice fixé) et de 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**.

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.

Extensions tableaux
Manipulations de tableaux

Sections non régulières

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**.

Exemples :

```
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)**.

5.3 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**.

```
integer, parameter      :: n=5,m=6
integer, dimension(n,m) :: t
t = 0
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 n'étant pas connue, 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*).

Tableau en argument d'une procédure

Reprise de l'exemple précédent :

```
integer, parameter      :: n=5,m=6
integer, dimension(n,m) :: t
t = 0
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 2^{e}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.

Tableau en argument d'une procédure

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 :

```
integer, dimension(5,6) :: t
  interface     !<<<<<<<<<<<<<<<<<<|
    subroutine sp(t)                |
      integer, dimension(:,:) :: t  |
    end subroutine sp               |
  end interface !<<<<<<<<<<<<<<<<<<|
t = 0 ; call sp(t)
end

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.

5.4 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...

$\Longrightarrow$ **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 ci-dessous.

Exemple :

dans 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.

Section de tableau non contiguë en argument d'une procédure


```
program tab_sect
 implicit none
 real, dimension(100,100) :: a1=1.0, a2
 real                     :: c1,c2,c3
 integer                  :: i
 interface  !-------------------------!
   subroutine sub1(x)                !
     real, dimension(:,:) :: x       !
   end subroutine sub1               !
 end interface !---------------------!
 a1(3,2:5) = (/ 3.0,4.0,5.0,6.0 /) ; a2 = a1
 call cpu_time(time=c1) ! <== Fortran 95 only !
 do i=1,1000
   call sub1(a1(3:70,2:50))! Section non contiguë
 enddo
 call cpu_time(time=c2)
 do i=1,1000
   call sub2(a2)          ! <== Tout le tableau
 enddo
 call cpu_time(time=c3)
 print *,"Durée_sub1:",c2-c1,",durée_sub2:",c3-c2
end program tab_sect

subroutine sub1(x)
 real, dimension(:,:) :: x
 x = x * 1.002
end subroutine sub1
```

Section de tableau non contiguë en argument d'une procédure

```
subroutine sub2(x)
 real, dimension(100,100) :: x
 x(3:70,2:50) = x(3:70,2:50) * 1.002
end subroutine sub2
```

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.*

5.5 Fonctions intrinsèques tableaux

5.5.1 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.

Fonctions intrinsèques tableaux : interrogation

Exemples

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

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

Fonctions intrinsèques tableaux : interrogation


```
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 un scalaire.

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

DIM=i $\Longrightarrow$ 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**.

Exemples :

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

Fonctions intrinsèques tableaux : interrogation

Exemples :

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

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

```
MAXLOC(array=A, mask=A.LT.5)  ⟹  (/ 2,2 /)
MINLOC(array=A, mask=A.GT.5)  ⟹  (/ 3,3 /)
MINLOC(array=A, mask=A>8 )    ⟹  résultat imprévisible

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 **`(/`** $d_1, d_2, \ldots, d_{i-1}, d_{i+1}, \ldots, d_n$ **`/)`**

5.5.2 Réduction (`all, any, count, sum,...`)

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` $\Longrightarrow$ la fonction travaille globalement sur cet indice (c.-à-d. un vecteur) pour chaque valeur fixée dans les autres dimensions.

Soient **A** $\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$ et **B** $\begin{pmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{pmatrix}$

- Réduction globale :
  `ALL(A.NE.B)` $\Longrightarrow$ **`.false.`**
- Réduction par colonne :
  `ALL(A.NE.B, dim=1)` $\Longrightarrow$
  `(/ .true.,.false.,.false. /)`
- Réduction par ligne :
  `ALL(A.NE.B, dim=2)` $\Longrightarrow$ **`(/ .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)))...`

Fonctions intrinsèques tableaux : réduction

ANY(mask[,dim])

Soient **A** $\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$ et **B** $\begin{pmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{pmatrix}$

- Réduction globale :

  ANY(A/=B) $\Longrightarrow$ **.true.**
- Réduction par colonne :

  ANY(A/=B, dim=1) $\Longrightarrow$ **(/ .true.,.false.,.true. /)**
- Réduction par ligne :

  ANY(A/=B, dim=2) $\Longrightarrow$ **(/ .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)))...

Fonctions intrinsèques tableaux : réduction

`COUNT(mask[,dim])`

`COUNT((/ .true.,.false.,.true. /))` $\Longrightarrow$ **`2`**

Soient **A** $\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$ et **B** $\begin{pmatrix} 0 & 3 & 5 \\ 7 & 4 & 8 \end{pmatrix}$

A/=B $\Longrightarrow$ $\begin{pmatrix} T & F & F \\ T & F & T \end{pmatrix}$

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

Fonctions intrinsèques tableaux : réduction

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

MINVAL((/ 1,4,9 /)) $\Longrightarrow$ **1**
MAXVAL((/ 1,4,9 /)) $\Longrightarrow$ **9**

Soit **A** $\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$

MINVAL(A,dim=1) $\Longrightarrow$ **(/ 1,3,5 /)**
MINVAL(A,dim=2) $\Longrightarrow$ **(/ 1,2 /)**
MAXVAL(A,dim=1) $\Longrightarrow$ **(/ 2,4,6 /)**
MAXVAL(A,dim=2) $\Longrightarrow$ **(/ 5,6 /)**
MINVAL(A,dim=1,mask=A>1) $\Longrightarrow$ **(/ 2,3,5 /)**
MINVAL(A,dim=2,mask=A>3) $\Longrightarrow$ **(/ 5,4 /)**
MAXVAL(A,dim=1,mask=A<6) $\Longrightarrow$ **(/ 2,4,5 /)**
MAXVAL(A,dim=2,mask=A<3) $\Longrightarrow$ **(/ 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.

Fonctions intrinsèques tableaux : réduction


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

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

Soit **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.

5.5.3 Multiplication (`matmul, dot_product,...`)

Il existe deux fonctions 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.

Exemples :

Soient les vecteurs **v1** `=(/ 2,-3,-1 /)` et **v2** `=(/ 6,3,3 /)` :

DOT_PRODUCT(v1,v2) $\Longrightarrow$ **0**

Soient la matrice **A** $\begin{pmatrix} 3 & -6 & -1 \\ 2 & 3 & 1 \\ -1 & -2 & 4 \end{pmatrix}$ et le vecteur **V** $\begin{pmatrix} 2 \\ -4 \\ 1 \end{pmatrix}$

MATMUL(A,V) $\Longrightarrow \begin{pmatrix} 29 \\ -7 \\ 10 \end{pmatrix}$

Fonctions intrinsèques tableaux : multiplication

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	$\Longrightarrow$	**sum(v1*v2)**
v1 de type complexe	$\Longrightarrow$	**sum(conjg(v1)*v2)**
v1 et **v2** de type logique	$\Longrightarrow$	**any(v1.and.v2)**

c = MATMUL(a,b)

Si **profil(a)=(/ n,p /)** et **profil(b)=(/ p,q /)** $\Longrightarrow$ **profil(c) = (/ n,q /))** $c_{i,j}$**=sum(a(i,:)*b(:,j)))**

Si **profil(a)=(/ p /)** et **profil(b)=(/ p,q /)** $\Longrightarrow$ **profil(c) = (/ q /))** c_j**=sum(a*b(:,j)))**

Si **profil(a)=(/ n,p /)** et **profil(b)=(/ p /)** $\Longrightarrow$ **profil(c) = (/ n /))** c_i**=sum(a(i,:)*b))**

Si **a** et **b** de type logique $\Longrightarrow$ On remplace **sum** par **any** et * par **.and.** dans les formules précédentes

5.5.4 Construction/transformation (`reshape, cshift, pack, spread, transpose,...`)

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

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

Exemples :

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

a pour valeur le tableau $\begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$

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

a pour valeur le tableau $\begin{pmatrix} T & F & F & T \\ F & T & F & T \\ F & F & T & T \\ F & F & F & T \end{pmatrix}$

Fonctions intrinsèques tableaux : construction et transformation

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 107).

Exemple :

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

a pour valeur le tableau $\begin{pmatrix} T & F & F & F \\ F & T & F & F \\ F & F & T & F \\ T & T & T & T \end{pmatrix}$

Fonctions intrinsèques tableaux : construction et transformation

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**

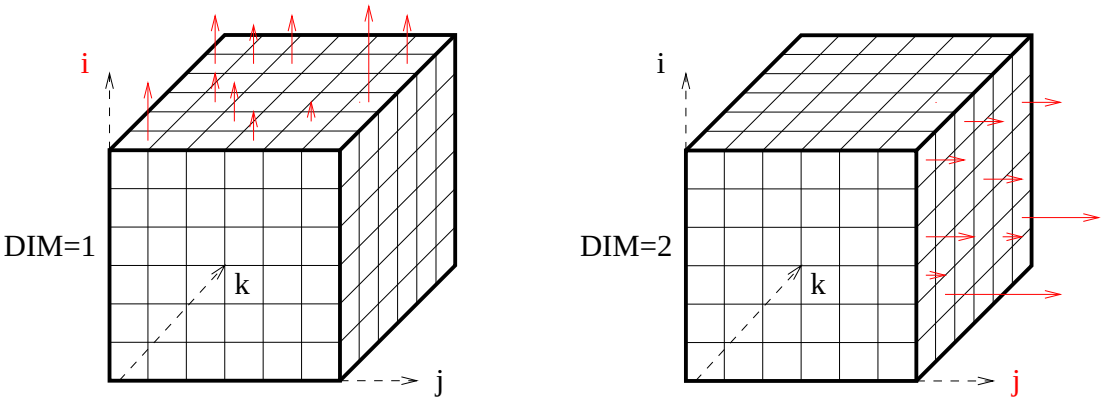
```
integer, dimension(6) :: v = (/ 1,2,3,4,5,6 /)
integer, dimension(6) :: w1,w2
w1 = CSHIFT(v, shift=2)
w2 = CSHIFT(v, shift=-2)
print *,w1(:)
print *,w2(:)
```

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

et **w2 = (/ 5,6,1,2,3,4 /)**

SHIFT > 0 $\Longrightarrow$ décalage vers les indices décroissants

SHIFT < 0 $\Longrightarrow$ 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 (/ $d_1, d_2, \ldots, d_n$ /), l'argument `shift` doit être un scalaire ou un tableau d'entiers de rang **n-1** et de profil (/ $d_1, d_2, \ldots, d_{dim-1}, d_{dim+1}, \ldots, d_n$ /).*

Fonctions intrinsèques tableaux : construction et transformation

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)
```

vaut

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

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

vaut

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

Fonctions intrinsèques tableaux : construction et transformation

Exemple : 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

remplacer **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.

Fonctions intrinsèques tableaux : construction et transformation

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 $\Longrightarrow$ décalage vers les indices décroissants

SHIFT < 0 $\Longrightarrow$ 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

Fonctions intrinsèques tableaux : construction et transformation

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 =(/ 4,4 /))
t2_car=EOSHIFT(array =t1_car,                  &
               shift =3,                       &
             boundary =(/ (achar(i),i=113,116) /))
```

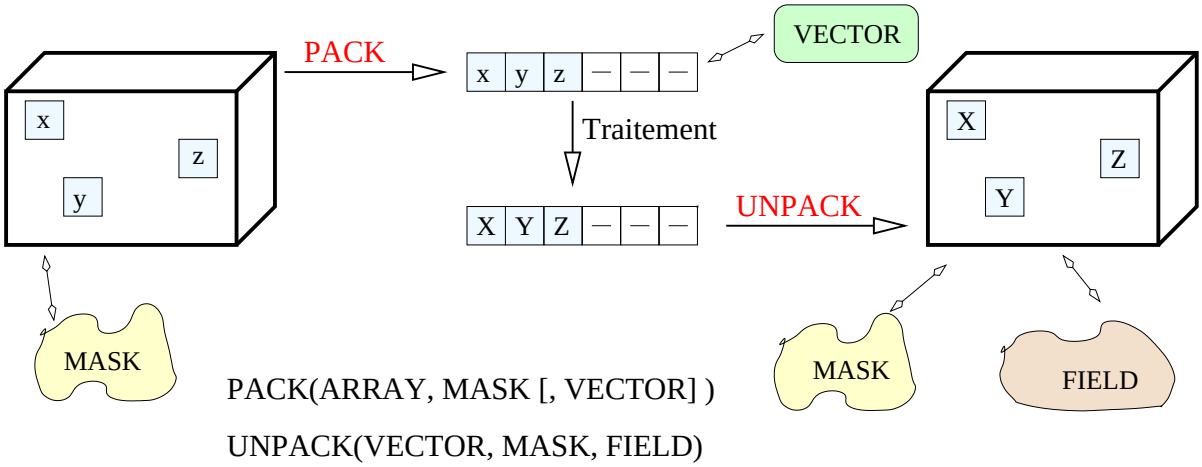
On obtient :

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

Fonctions intrinsèques tableaux : construction et transformation

PACK(array,mask[,vector])

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



PACK(ARRAY, MASK [, VECTOR])

UNPACK(VECTOR, MASK, FIELD)

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) /))
```

On obtient : `(/ 9,6 /)`

`(/ 9,6,27,64,125 /)`

Fonctions intrinsèques tableaux : construction et transformation

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 =(/ 3,3 /))
fld= UNPACK(v2, mask=m, field=0)
m  = CSHIFT(m,  shift=(/ -1,1,0 /), dim=2)
a  = UNPACK(v,  mask=m, field=fld)
```

On obtient :

$$\text{m} \begin{pmatrix} T & F & F \\ F & T & F \\ F & F & T \end{pmatrix} \quad \text{fld} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

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

Fonctions intrinsèques tableaux : construction et transformation

Exemple : 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=(/ n,n /))
!--Compression (éléments tri-diagonaux)
v=pack(A,mask=m)
!--Traitement des éléments tridiagonaux
!--compressés
v = v+1. ; . . . . .
!--Décompression après traitement
A=unpack(v,mask=m,field=A)
!--Impression
do i=1,size(A,1)
  print '(10(1x,f7.5))', A(i,:)
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**.

Exemple : duplication selon les lignes par un facteur 3 :

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

On obtient le résultat suivant :

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

Exemple : dilatation/expansion d'un vecteur par un facteur 4 :

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

On obtient le résultat suivant :

```
 (/ 4 4 4 4 8 8 8 8 2 2 2 2 /)
```

Fonctions intrinsèques tableaux : construction et transformation

MERGE(tsource,fsource,mask)

Fusion de deux tableaux sous le contrôle d'un masque.

Exemple :

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

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

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

TRANSPOSE(matrix)

Cette fonction permet de transposer la matrice passée en argument.

Exemples :

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

On obtient :

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

5.6 Instruction et bloc WHERE

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

Forme générale :

```
[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
```

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 : $\Longrightarrow$ `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 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**.

Extensions tableaux
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).

5.7 Expressions d'initialisation autorisées

- constructeur de vecteur (avec boucles implicites) :

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

- constructeur de structure :

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

- fonctions intrinsèques élémentaires si arguments d'appel et valeur retournée sont de type **INTEGER/CHARACTER** :

```
integer,parameter:: n=12**4
integer(kind=2)  :: l=int(n,kind=2)
integer          :: k=index(str,"IDRIS")
real :: x=real(n) !**INTERDIT** réel en retour
```

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

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

- certaines fonctions de transformation : **REPEAT**, **RESHAPE**, **TRIM**, **SELECTED_INT_KIND**, **SELECTED_REAL_KIND**, **TRANSFER**, **NULL**

```
integer, dimension(4,2) :: t = &
         reshape((/(i,i=1,8)/),(/4,2/))
real, pointer :: p=>null()
```

5.8 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.)`
3^{e}ligne de **M**	`M(3,:)`
2^{e}colonne de **M**	`M(:,2)`
$\sum_{1\le i\le 3} \sum_{2\le j\le 4} M_{ij}$	`SUM(M(1:3,2:4))`

Extensions tableaux
Exemples

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)`

Extensions tableaux
Exemples

Page réservée pour vos notes personnelles...

6 Gestion mémoire

⇒ Tableaux automatiques

⇒ Tableaux dynamiques (ALLOCATABLE, profil différé)

6.1 Tableaux automatiques

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 entières non constantes.
Ces tableaux sont appelés **tableaux automatiques** ; c'est le cas des tableaux **C** et **V** de l'exemple suivant.

```
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.

6.2 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.

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

Remarques :

- 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.
- Le retour d'une fonction et les arguments muets ne peuvent pas avoir l'attribut *allocatable*.
- Un tableau dynamique (*allocatable*) doit avoir été alloué avant de pouvoir être passé en argument d'appel d'une procédure (ce qui n'est pas le cas des tableaux avec l'attribut **pointer** — cf. paragraphe "Passage en argument de procédure" du chapitre suivant sur les pointeurs page 131). Dans la procédure, il sera considéré comme un simple tableau à profil implicite (sans l'attribut **ALLOCATABLE**) : inutile de chercher à l'y désallouer ou à le passer en argument de la fonction **ALLOCATED** !

- 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.
- Attention : en cas de problème au moment de l'allocation et en l'absence du paramètre **STAT=err**, 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 **err** qui est différente de zéro en cas de problème.

Page réservée pour vos notes personnelles...

7 Pointeurs

⇒ Définition, états d'un pointeur

⇒ Déclaration d'un pointeur

⇒ Symbole **=>**

⇒ Symbole **=** appliqué aux pointeurs

⇒ Allocation dynamique de mémoire

⇒ Fonction **`NULL()`** et instruction **`NULLIFY`**

⇒ Fonction intrinsèque **`ASSOCIATED`**

⇒ Situations à éviter

⇒ Déclaration de "tableaux de pointeurs"

⇒ Passage en argument de procédure

⇒ Pointeur, tableau à profil différé et **`COMMON`**

⇒ Liste chaînée : exemple

7.1 Définition, états d'un pointeur

Définition

En **C, Pascal** $\Longrightarrow$ variable contenant l'adresse d'objets

En **Fortran 90** $\Longrightarrow$ alias

États d'un pointeur

1. **Indéfini** $\Longrightarrow$ à sa déclaration
   en tête de programme
2. **Nul** $\Longrightarrow$ alias d'aucun objet
3. **Associé** $\Longrightarrow$ 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.*

7.2 Déclaration d'un pointeur

Attribut **`pointer`** spécifié lors de sa déclaration.

Exemples :

```
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 "tableau 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
```

7.3 Symbole =>

Cible : c'est un objet pointé.

Cet objet devra avoir l'attribut **`target`** lors de sa déclaration.

Exemple :

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

Le symbole => sert à *valoriser* un pointeur.

Il est binaire : **op1** => **op2**.

	Pointeur	Cible
op1	⊗	
op2	⊗	⊗

Pointeurs : symbole =>

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,

p1 => **p2** implique que **p1** prend l'état de **p2** :

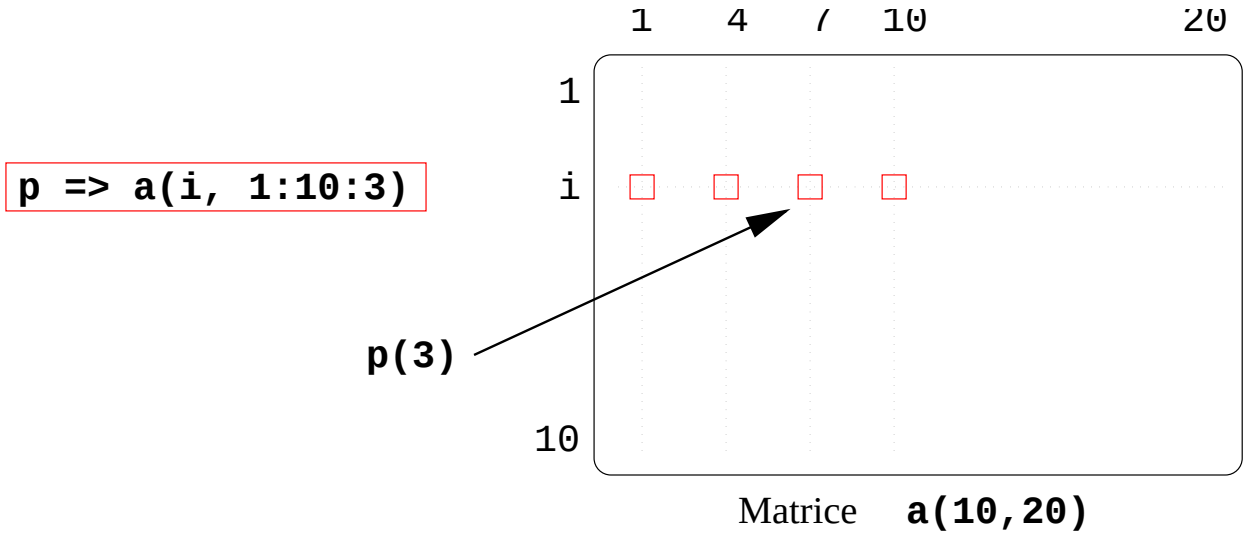
indéfini, nul ou associé à la même cible.

Pointeurs : symbole =>

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 /))
read(*, *) i
p => a(i, 1:10:3) ! p est maintenant un vecteur
print *, p(3)     ! de profil (/ 4 /)
end
```



7.4 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.

```
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
print *, ptr2
!--------------------------------------
a = reshape(source = (/ (i, i=1,9) /), &
            shape  = (/ 3,3 /))
p => a
q => b
q = p + 1.   ! ou  q = a + 1.
print *, b
end
```

Dans cet exemple, l'instruction : **ptr2 = ptr1**

est équivalente à : **i2 = i1**

7.5 Allocation dynamique de mémoire

Instruction **ALLOCATE** :

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

```
implicit none
integer, dimension(:,:), pointer :: p
integer                          :: n
read(*, *) n
ALLOCATE(p(n, n))
p = reshape((/ (i, i=1,n*n) /), (/ n,n /))
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 52 et 134).

7.6 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).

```
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`**.

7.7 Fonction intrinsèque ASSOCIATED

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

Syntaxe :

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*.

7.8 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.

Pointeurs
Situations à éviter

Exemple 2 :

```
implicit none
real, dimension(:,:), pointer :: p
integer                       :: n
read(*, *) n
allocate(p(n, 2*n))
p = 1.
nullify(p)
  . . .
```

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

7.9 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 :

```
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
 logical                        :: tri_termine
 integer                        :: i, eof
 print *,’Entrez le nombre de chaînes :’
 read(*, *) nbre_chaines
 allocate(tab_pointeurs(nbre_chaines))
 allocate(chaines(nbre_chaines))
```

Pointeurs
“Tableaux de pointeurs”

```
 do i=1,nbre_chaines
   print *,"Entrez une chaîne : "
   read(*, *) chaines(i)
   tab_pointeurs(i)%p => chaines(i)
 end do
 do
   tri_termine = .true.
   do i=1, nbre_chaines - 1
     if (tab_pointeurs(i  )%p(5:9) > &
         tab_pointeurs(i+1)%p(5:9)) then
       !--Permutation des deux associations----
       tab_pointeurs(i:i+1)   =   &
       tab_pointeurs(i+1:i:-1)
       !---------------------------------------
       tri_termine = .false.
     end if
   end do
   if (tri_termine) exit
 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.*

7.10 Pointeur passé 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. 5.4 page 76).
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`**.

Cible en argument d'une 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. Il s'agit dans tous les cas d'un passage d'argument classique par adresse. Si l'argument muet a l'attribut **`target`**, l'interface doit être explicite.

Attention à l'utilisation des pointeurs *globaux* ou *locaux permanents* *(***`save`***)* éventuellement associés dans la procédure à cette cible dans le cas où le compilateur aurait dû faire une copie *copy in–copy out* de l'argument d'appel (cf. chapitre 5 page 76)...

D'ailleurs, d'une façon générale, la norme ne garantit pas la conservation de l'association de pointeurs entre la cible passée en argument et celle correspondant à l'argument muet.

7.11 Pointeur, tableau à profil différé et COMMON : exemple

```
!real,allocatable,dimension(:,:)  :: P ! INTERDIT
 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).

7.12 Liste chaînée : exemple

```
module A   ! Petit exemple de liste chaînée en Fortran 95
  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
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) !=>Impress. de la dernière à la 1ère.
  call libere(debut)  !=>Libération totale de la liste.
end program liste
```

8 Interface de procédures et modules

- ⇒ Interface implicite : définition
- ⇒ Interface implicite : exemple
- ⇒ Arguments : attributs **`INTENT`** et **`OPTIONAL`**
- ⇒ Passage d'arguments par mot-clé
- ⇒ Interface explicite : procédure interne (**`CONTAINS`**)
- ⇒ Interface explicite : 5 cas possibles
- ⇒ Interface explicite : bloc interface
- ⇒ Interface explicite : ses apports
- ⇒ Interface explicite : module avec bloc interface (**`USE`**)
- ⇒ Interface explicite : module avec procédure
- ⇒ Cas d'interface explicite obligatoire
- ⇒ Argument de type procédural et bloc interface

8.1 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 $\Longrightarrow$ cf. exemple ci-après montrant quelques erreurs classiques non détectées à la compilation.

Exemple :

Utilisation d’un sous-progr. externe **`maxmin`** pour calculer les valeurs max. et min. d’un vecteur **`vect`** de taille **`n`** et optionnellement le rang **`rgmax`** de la valeur max. avec mise-à-jour de la variable de contrôle **`ctl`**.

8.2 Interface implicite : exemple

```
program inout
  implicit none
  integer, parameter :: n=100
  real,dimension(n)  :: v
  real               :: xv,y
  ..........
  call maxmin(v,n,vmax,vmin,ctl,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,ctl,rgmax)
!-->> Interversion de deux arguments
  call maxmin(v, n,vmax,vmin,rgmax,ctl)
!-->> "Oubli" de l’argument rgmax
  call maxmin(v, n,vmax,vmin,ctl)
!-->> Argument y en trop
  call maxmin(v, n,vmax,vmin,ctl,rgmax,y)
end program inout
subroutine maxmin(vect,n,v_max,v_min,ctl,rgmax)
  real,dimension(n)  :: vect
  ..........
  V=v_max+... !-Erreur: v_max en sortie seult.
  n=....      !-Erreur: n en entrée seulement
  ctl=99      !-Erreur: constante passée en arg.
  ..........
```

8.3 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 :
     - entrée seulement $\Longrightarrow$ **`INTENT(IN)`**,
     - sortie seulement $\Longrightarrow$ **`INTENT(OUT)`** : dans la procédure, l'argument muet doit être défini avant toute référence à cet argument,
     - mixte $\Longrightarrow$ **`INTENT(INOUT)`**,

```
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)  :: rgmax
. . . . .
if (present(rgmax)) then ...
```

Remarques à propos de l'attribut `INTENT`

- 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,
  - d'un argument ayant l'attribut **`POINTER`**.
- **`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.

8.4 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.

Règle : 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.

Si **`rgmax`** a l'attribut **`OPTIONAL`**

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

Exemple :

appel du 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.

8.5 Interface explicite : procédure interne (CONTAINS)

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

Interface “explicite” Procédure interne (CONTAINS)

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.

Notes sur les procédures internes et l’**`IMPLICIT NONE`** :

- il n’est pas possible d’imbriquer les procédures internes ; le **`CONTAINS`** est à un seul niveau ;
- les procédures internes et la procédure les contenant forment une même et unique *scoping unit* ;
- l’**`IMPLICIT NONE`** d’une procédure se transmet à l’ensemble de la *scoping unit* et donc aux procédures internes ;
- l’**`IMPLICIT NONE`** de la partie "data" (*specification part*) d’un module n’est pas exportable via **`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 d’utiliser l’**`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.

8.6 Interface explicite : 5 cas possibles

1. procédures intrinsèques (Fortran 77 et Fortran 95),
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.

8.7 Interface explicite : bloc interface

Pour éviter les inconvénients de la procédure interne tout en conservant la fiabilité de l’interface “*explicite*”, Fortran 90 offre la solution :

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 :

Voici le même exemple avec **procédure externe** et **bloc interface** :

Interface “explicite”
Bloc interface


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

Interface “explicite”
Bloc interface

```
 v_max = MAXVAL(vect)
 v_min = MINVAL(vect)
 ctl = 1
 if(present(rgmax)) then
   rgmax = MAXLOC(vect, DIM=1)
   ctl = 2
 endif
 print *,’Taille vecteur via size :’,SIZE(vect)
 print *,’Profil vecteur via shape:’,SHAPE(vect)
end subroutine maxmin
```

8.8 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.

8.9 Interface explicite : module et bloc interface (USE)

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.

Voici deux exemples d’utilisation du module pour réaliser une interface “explicite” :

- module avec bloc interface,
- module avec procédure (solution la plus sûre).

Interface “explicite”
Module avec bloc interface


```
module bi_maxmin
interface
 subroutine maxmin(vect,v_max,v_min,ctl,rgmax)
   real,dimension(:), intent(in)   :: vect
   real,              intent(out)  :: v_max,v_min
   integer, optional, intent(out)  :: rgmax
   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,ctl=0
  real,dimension(n)  :: v=(/ 1.,2.,40.,3.,4. /)
  real               :: vmax,vmin
  .......
  call maxmin(v, vmax, vmin, ctl, rgmax)
  .......
end program inout
```

8.10 Interface explicite : module avec procédure

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

Après compilation séparée du module, on l’utilisera par :

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

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

8.11 Cas d'interface explicite obligatoire

Il est des cas où l'interface d'appel doit être "explicite". Il en existe **10** :

- 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 **`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 : 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
```

Cas d'interface "explicite" obligatoire

```
  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

program ex2
  use M1
  implicit none
  real, dimension(:), pointer :: ptr
  integer                     :: i
  real, dimension(100)        :: t_in
  real, dimension(102)        :: t_out
  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
  !------ Appel fonction retournant une chaîne
  print *, f_c(str="abcdef")
end program ex2
```

Cas d'interface "explicite" obligatoire

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. Voici un 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
 real x,y      ! ********   spécifié au niveau de
 .  .  .  .    !            f dans le module A
 y=f(x)
 .  .  .  .
end program pg1
```

Cette interdiction est justifiée par la volonté d'éviter des redondances inutiles ou même contradictoires !

8.12 Argument de type procédural et bloc interface

```
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(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

program P
use fct
  implicit none
  real                  :: x
  real,dimension(10) :: t
    . . .
  x = myfonc(t, f1) ! avec arg. d'appel f1
  x = myfonc(t, 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 essayer de *voir* le bloc interface par *use association*.

9 Interface générique

⇒ Introduction

⇒ Exemple avec **`module procedure`**

⇒ Exemple : contrôle de procédure F77

9.1 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.

9.2 Exemple avec `module procedure`

Définition d'une fonction générique **`maxmin`** s'appliquant aux vecteurs qu'ils soient de type réel ou de type entier $\Longrightarrow$ 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-progr. 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`**.

Interface générique
Exemple avec `module procedure`

```
module big_maxmin
 interface maxmin
   module procedure rmaxmin,imaxmin !<<<<<<
 end interface maxmin !<-- F95 only
contains
subroutine rmaxmin(vect,v_max,v_min,ctl,rgmax)
 implicit none
 real,dimension(:), intent(in)    :: vect
 real,              intent(out)   :: v_max,v_min
 integer, optional, intent(out)   :: rgmax
 integer,           intent(inout) :: ctl
 v_max=MAXVAL(vect); v_min=MINVAL(vect); ctl=1
 if(present(rgmax)) then !-- fonction logique
   rgmax=MAXLOC(vect, DIM=1); ctl=2
 endif
end subroutine rmaxmin
!----------------------
subroutine imaxmin(vect,v_max,v_min,ctl,rgmax)
 implicit none
 integer,dimension(:), intent(in) :: vect
 integer,           intent(out)   :: v_max,v_min
 integer, optional, intent(out)   :: rgmax
 integer,           intent(inout) :: ctl
 v_max=MAXVAL(vect); v_min=MINVAL(vect); ctl=1
 if(present(rgmax)) then !-- fonction logique
   rgmax=MAXLOC(vect, DIM=1); ctl=2
 endif
end subroutine imaxmin
end module big_maxmin
```

Interface générique
Exemple avec `module procedure`

Voici le programme utilisant ce module :

```
program inout
  USE big_maxmin  !<<<--Accès au bloc interface
  implicit none   !     et aux procédures
  integer, parameter :: n=5
  real,dimension(n)  :: v=(/ 1.,2.,40.,3.,4. /)
  . . .
  call maxmin(v, vmax, vmin, ctl, rgmax)
  . . .
  call sp1(n+2)
  . . .
end program inout
!
subroutine sp1(k)
  USE big_maxmin  !<<<--Accès au bloc interface
  implicit none   !     et aux procédures
  integer, dimension(k) :: v_auto
  . . .
  call maxmin(v_auto, vmax, vmin, ctl, rgmax)
  . . .
end subroutine sp1
```

Interface générique
Exemple avec `module procedure`

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 :

```
interface maxmin
 MODULE PROCEDURE imaxmin
 subroutine rmaxmin(vect,v_max,v_min,ctl,rgmax)
   real,dimension(:), intent(in)   :: vect
   real,              intent(out)  :: v_max,v_min
   integer, optional, intent(out)  :: rgmax
   integer,           intent(inout):: ctl
 end subroutine rmaxmin
end interface maxmin !<-- F95 only
```

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.

9.3 Exemple : contrôle de procédure F77

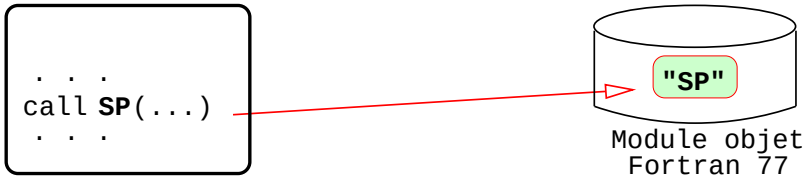


Schéma 1 : appel classique d’un sous-progr. **SP** contenu dans un *module objet* Fortran 77 en mode “interface implicite ” sans contrôle inter-procédural.

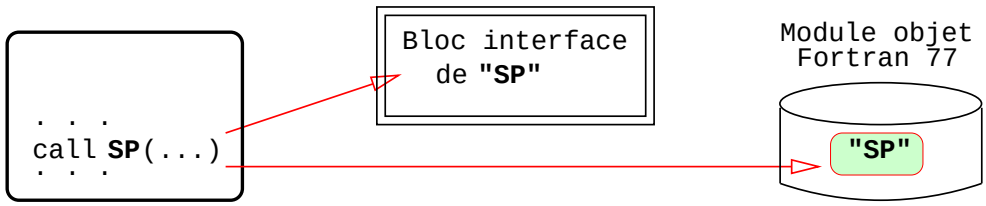


Schéma 2 : idem en contrôlant le passage d’arguments via un “**bloc interface**”.

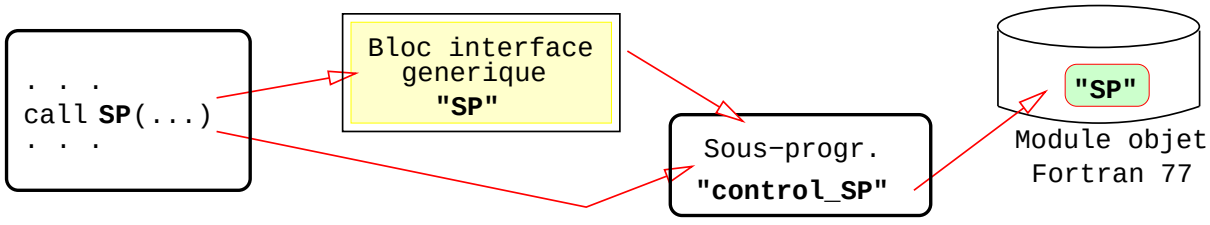


Schéma 3 : idem en utilisant un **bloc interface générique SP** appelant un sous-programme **`control_SP`** contrôlant l’appel de **SP** avec la notion d’arguments optionnels et de valeurs par défaut associées (cf. exemple ci-après).

Contrôle de procédure F77 via interface “générique et explicite”

```
  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
!-----------------------------------------------
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
```

Contrôle de procédure F77 via interface “générique et explicite”

Autre solution

```
module module_generic
interface SP                      !-----------------
 module procedure control_SP !Bloc interface SP
end interface SP                  !-----------------
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
 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
!-----------------------------------------------
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
```

Contrôle de procédure F77 via interface “générique et explicite”

Page réservée pour vos notes personnelles...

10 Surcharge ou création d'opérateurs

⇒ Introduction

⇒ Interface **`operator`**

⇒ Interface **`assignment`**

10.1 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.

Exemples :

```
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=(/ 3,3 /))
b = reshape((/ (i**3,i=1,9) /), shape=(/ 3,3 /))
    ...
c = b
u = v
l = a == b
if (a == b)... ! Incorrect POURQUOI ?
l = a < b
c = a - b
c = a * b
    ...
```

10.2 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 +** :

```
module matrix
  implicit none
  type OBJ_MAT
    integer                          :: n,m
    real, dimension(:,:), pointer :: ptr_mat
  end type OBJ_MAT
  interface operator(+)
    module procedure add
  end interface
contains
  function add(a,b)
    type(OBJ_MAT), intent(in)   :: a,b
    type(OBJ_MAT)               :: add
    integer(kind=2)             :: err
    add%n = a%n; add%m = a%m
    allocate(add%ptr_mat(add%n,add%m),stat=err)
    if (err /= 0) then
      print *,'Erreur allocation' ; stop 4
    endif
    add%ptr_mat = a%ptr_mat + b%ptr_mat
  end function add
end module matrix
```

```
program appel
 use matrix
 implicit none
 integer       :: err, 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%ptr_mat(n,m), stat=err)
 if (err /= 0) then
   print *,'Erreur allocation matrice u'; stop 4
 endif
 allocate(v%ptr_mat(n,m), stat=err)
 if (err /= 0) then
   print *,'Erreur allocation matrice v'; stop 4
 endif
 u%ptr_mat = reshape( &
 (/ ((real(i+j),i=1,n),j=1,m) /),shape=(/ n,m /))
 v%ptr_mat = reshape( &
 (/ ((real(i*j),i=1,n),j=1,m) /),shape=(/ n,m /))
 w = u + v ! <<<<<<<<<<<<<
 do i=1,w%n
   print *, w%ptr_mat(i,:)
 end do
end program appel
```

10.3 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 :

```
module matrix
  implicit none
  integer(kind=2), private        :: err

  type OBJ_MAT
    integer                      :: n,m
    real, dimension(:,:), pointer :: ptr_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
contains
  function add(a,b)
    . . .
  end function add
```

```
  function trans(a)
! Fonction associée à l'opérateur .tr.
    type(OBJ_MAT), intent(in)   :: a
    type(OBJ_MAT)               :: trans
    trans%n = a%m;trans%m = a%n
    allocate(trans%ptr_mat(trans%n,trans%m), &
             stat=err)
    if (err /= 0) then
      print *,'Erreur allocation'
      stop 4
    endif
    trans%ptr_mat = transpose(a%ptr_mat)
  end function trans

  subroutine taille_mat(i,a)
! Sous-programme associé à l'affectation (=)
    integer,       intent(out) :: i
    type(OBJ_MAT), intent(in)  :: a
    i = a%n*a%m
  end subroutine taille_mat
end module matrix
```

Surcharge d'opérateurs
Interface assignment

```
program appel
  use matrix
  implicit none
  type(OBJ_MAT) :: u, v, w, t
  integer       :: err, taille_u, taille_v, n, m
     ...
  read *, n, m
  allocate(u%ptr_mat(n,m), &
           v%ptr_mat(n,m), &
           w%ptr_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 1er argument (opérande de gauche) doit avoir l'attribut **`intent(out)`** ou **`intent(inout)`** et le 2e (opér. de droite), l'attribut **`intent(in)`**.
- En l'absence du paramètre **`stat=`** de l'**`allocate`** et en cas d'erreur, une action standard arrête le programme avec *traceback* et fichier *core* éventuel.
- Les symboles => (*pointer assignment symbol*) et % (référence à une composante de structure) ne peuvent être surchargés.

11 Contrôle de visibilité, concept d'encapsulation et gestion de zones dynamiques

⇒ Introduction
⇒ Instruction **PRIVATE** et **PUBLIC**
⇒ Attribut **PRIVATE** et **PUBLIC**
⇒ Type dérivé "semi-privé"
⇒ Exemple avec gestion de zones dynamiques inaccessibles en retour de fonction
⇒ Paramètre **ONLY** de l'instruction **USE**

Contrôle visibilité, encapsulation : introduction

11.1 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.

11.2 Instruction 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.

Exemples :

```
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
```

11.3 Attribut 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.

Exemples :

```
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 les variables **x**, **y** et **z** il serait préférable de coder :

```
real, public :: x
real         :: y, z
```

11.4 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**.

Contrôle visibilité, encapsulation : type dérivé “semi-privé”

Exemples :

```
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
```

11.5 Exemple avec gestion de zones dynamiques inaccessibles en retour de fonction

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

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

```
module matrix
  integer           :: nb_lignes, nb_col
  integer, private :: err
  !
  type OBJ_MAT
    private
    integer                          :: n=0,m=0 !
    real,dimension(:,:),pointer :: &        ! F95 only
          ptr_mat => NULL()                  ! --------
  end type OBJ_MAT
  !
  private :: valorisation,add,taille_mat,trans
!--------------------------------
  interface operator(+)
     module procedure add
  end interface
```

Contrôle visibilité, encapsulation : exemple avec zones dynamiques

```
!---------------------------------
  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 (.not.associated(a%ptr_mat)) then
      allocate(a%ptr_mat(nb_lignes,nb_col), &
               stat=err)
      if (err /= 0) then
        print *,"Impossible de créer &
                 &l'objet indiqué."
        stop 4
      endif
      a%n = nb_lignes; a%m = nb_col
    endif
    a%ptr_mat = reshape(source = t,              &
                        shape  = (/ a%n, a%m /))
  end subroutine valorisation
```

Contrôle visibilité, encapsulation : exemple avec zones dynamiques


```
!--------------------------------
  subroutine poubelle(a)
    type(OBJ_MAT), intent(inout) :: a
    if (associated(a%ptr_mat)) then
      a%n = 0; a%m = 0
      deallocate(a%ptr_mat)
    endif
  end subroutine poubelle
!--------------------------------
  function add(a,b)
    type(OBJ_MAT), intent(in)   :: a,b
    type(OBJ_MAT)               :: add
    allocate(add%ptr_mat(a%n,a%m),stat=err)
    if (err /= 0) then
      print *,"Impossible de créer &
              &l'objet indiqué."
      stop 4
    endif
    add%n = a%n; add%m = a%m
    add%ptr_mat = a%ptr_mat + b%ptr_mat
  end function add
!--------------------------------
  subroutine taille_mat(i,a)
    integer,       intent(out) :: i
    type(OBJ_MAT), intent(in)  :: a
    i = a%n*a%m
  end subroutine taille_mat
```

Contrôle visibilité, encapsulation : exemple avec zones dynamiques

```
!--------------------------------
  function trans(a)
    type(OBJ_MAT), intent(in)   :: a
    type(OBJ_MAT)               :: trans
    allocate(trans%ptr_mat(a%m, a%n), stat=err)
    if (err /= 0) then
      print *,"Impossible de créer &
              &l'objet indiqué."
      stop 4
    endif
    trans%n = a%m
    trans%m = a%n
    trans%ptr_mat = transpose(a%ptr_mat)
  end function trans
!--------------------------------
  subroutine imp(a)
    type(OBJ_MAT), intent(in)  :: a
    integer(kind=2)            :: i
    do i=1,size(a%ptr_mat,1)
!   do i=1,a%n
      print *,a%ptr_mat(i,:)
    enddo
    print *,'-----------------------'
  end subroutine imp
!--------------------------------
end module matrix
```

Contrôle visibilité, encapsulation : exemple avec zones dynamiques

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_col
  u=(/ ((real(i+j),i=1,nb_lignes),j=1,nb_col) /)
  v=(/ ((real(i*j),i=1,nb_lignes),j=1,nb_col) /)
  . . . . . .
  u=v
  do i=1,1000;  ...;  w = u + v;  end do
  . . . . . .

  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
```

Contrôle visibilité, encapsulation : exemple avec zones dynamiques

Gestion des zones anonymes dormantes

- **Que se passe-t'il lors de l'affectation `u = v` ?**

  Lors de l'affectation entre deux structures le compilateur réalise effectivement des affectations entre les composantes sauf pour celles qui ont l'attribut **pointeur** pour lesquelles il réalise une **association**.

  Dans notre exemple il effectue donc :

  `u%ptr_mat => v%ptr_mat` ; **`u%ptr_mat`** prend donc l'état de **`v%ptr_mat`** c.-à-d. associé à la même cible. De ce fait la zone mémoire *anonyme* qui était auparavant associée à **`u%ptr_mat`** ne peut plus être référencée et devient donc une zone dormante encombrante et inutile ! De plus, **`u`** et **`v`** ne sont plus indépendants.

- **Que faudrait-il faire ?**

  Dans ce cas, il est préférable de surcharger le symbole d'affectation en gardant la maîtrise complète des opérations à effectuer. Dans le module **`matrix`**, on rajoute donc la procédure **`affect`** au niveau du bloc interface **`interface assignment(=)`** et on écrit un sous-programme **`affect`** du type de celui dont vous avez la liste sur la page suivante. La solution adoptée élimine le problème de la zone *anonyme dormante* et évite les problèmes liés à la non-initialisation éventuelle des variables de l'affectation.

Contrôle visibilité, encapsulation : exemple avec zones dynamiques


```
module matrix !<== Solution avec redéfinition de
  . . . . .   !    l'affectation
!-------------------------------
  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.associated(b%ptr_mat))     &
    stop "Erreur : membre de droite de  &
          &l'affectation non initialisé"
    if (associated(a%ptr_mat)) then
      if(any(shape(a%ptr_mat) /= shape(b%ptr_mat))) &
      stop "Erreur : affect. matrices non conformantes"
    else
      allocate(a%ptr_mat(b%n,b%m), stat=err)
      if (err /= 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=err.
    end if
    a%n = b%n ; a%m = b%m
    a%ptr_mat = b%ptr_mat
  end subroutine affect
  . . . . .
```

Contrôle visibilité, encapsulation : exemple avec zones dynamiques

Question : que se passe-t'il alors dans les 2 cas suivants ?

1. `w = u + v + t` ou bien, `W = .tr.(u + v)`
2. `do i=1,n; ...; w = u + v; ...; end do`

L'évaluation de ces expressions implique de multiples appels aux fonctions **`add`** et/ou **`trans`**. Les tableaux **`add%ptr_mat`** (ou **`trans%ptr_mat`**) alloués dynamiquement à chaque appel de ces fonctions deviennent inaccessibles ; ce sont des zones *anonymes dormantes*.

$\Longrightarrow$ **Risque de saturation mémoire !**

La libération automatique de ces *zones dormantes* n'étant pas prise en charge par le compilateur, c'est au programmeur d'assurer la fonction *ramasse-miettes*. Pour ce faire nous allons ajouter au type **`OBJ_MAT`** une composante supplémentaire permettant de savoir si un objet de ce type a été "**créé par une fonction**". De type logique, cette composante sera *vraie* si la création est faite dans une fonction comme **`add`** ou **`trans`** et *fausse* dans les autres cas. Là où c'est nécessaire (procédures **`add`**, **`trans`**, **`affect`** et **`imp`**) on ajoute alors la libération de la *zone anonyme dormante* si cette composante est *vraie*.

Contrôle visibilité, encapsulation : exemple avec zones dynamiques

Voici un extrait de la solution adoptée en version Fortran 95.

```
module matrix  ! <== Solution avec "ramasse-miettes"
  . . . . .    !     (en Fortran 95)
  . . . . .
  type OBJ_MAT
    private
    logical :: CreeParFonction=.false.
    integer :: n=0, m=0
    real,dimension(:,:),pointer :: ptr_mat => NULL()
  end type OBJ_MAT
  . . . . .
  . . . . .
  private :: ramasse_miettes
contains
  subroutine ramasse_miettes(a)
    type(OBJ_MAT), intent(in) :: a
    type(OBJ_MAT)             :: temp
    !
    temp%ptr_mat => a%ptr_mat
    call poubelle(temp)
  end subroutine ramasse_miettes
  . . . . .
  . . . . .
  function add(a,b)
    . . . . .
    add%CreeParFonction = .true.
    if (a%CreeParFonction) call ramasse_miettes(a)
    if (b%CreeParFonction) call ramasse_miettes(b)
  end function add
  . . . . .
```

Contrôle visibilité, encapsulation : exemple avec zones dynamiques

```
. . . . .
function trans(a)
  . . . . .
  trans%CreeParFonction = .true.
  if (a%CreeParFonction) call ramasse_miettes(a)
end function trans

function imp(a)
  . . . . .
  if (a%CreeParFonction) call ramasse_miettes(a)
end function imp

subroutine affect(a,b)!<=== Cf. vers. modifiée page
  . . . . .           !     (***)
  if (b%CreeParFonction) call ramasse_miettes(b)
end subroutine affect
. . . . .
```

Attention *: l'appel à* `ramasse_miettes(a)` *ne pourrait être remplacé par* `call poubelle(a)` *ou même* `deallocate(a%ptr_mat)` *car ce faisant, on modifierait la composante pointeur de* `a` *(son descripteur en fait) qui est protégé par la vocation* `INTENT(in)` *obligatoire pour les arguments muets d'une fonction de surcharge.*

(***) ⟹ page 185

11.6 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**.

```
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
```

Contrôle de visibilité : paramètre ONLY de l'instruction USE

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`**.

Exemple :

```
use m, mon_t2=>t2, mon_f=>f
use m, only : mon_t2=>t2, mon_f=>f
```

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

12 Procédures récursives

⇒ Clauses **`RESULT`** et **`RECURSIVE`**

⇒ Exemple : suite de Fibonacci

12.1 Clauses RESULT et RECURSIVE

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

Définition d'une procédure récursive :

```
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.

12.2 Exemple : suite de Fibonacci

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

```
recursive function fibonacci(n) result(fibo)
  integer, intent(in):: n
  integer             :: fibo
  integer, save       :: penult, antepenult
  !-----------------------------------------
  if (n <= 1) then  !--> Test d'arrêt
                    !    On peut dépiler
    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
```

Procédures récursives
Exemple

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 interdite par la norme Fortran 95 ! En effet, dans une expression, l'appel d'une fonction n'a pas le droit de modifier une entité (ici la variable locale **`antepenult`** avec l'attribut **SAVE**) intervenant dans cette expression. De plus, 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. chap. 7.12 page 134

13 Nouveautés sur les E/S

- ⇒ **OPEN** (**status, position, action, ...**)
- ⇒ **INQUIRE** (**recl, action, iolength,...**)
- ⇒ Entrées-sorties sur les fichiers texte (**advance='no'**)
- ⇒ Instruction **NAMELIST**
- ⇒ Spécification de format minimum

13.1 OPEN(status, position, action, ...)

STATUS :

- **REPLACE** : si le fichier n'existe pas, il sera créé, sinon il sera détruit et un fichier de même nom sera créé.

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 **open**) modifier certaines caractéristiques du fichier tout en restant positionné (valeur par défaut). Très limitatif et dépendant du constructeur !

PAD :

- **YES** : des enregistrements lus avec format sont complétés avec des blancs (*padding*) si *Input list* $>$ *Record length*
  (valeur par défaut).
- **NO** : pas de *padding*.

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).

DELIM :

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 la quote " sera utilisée.
- **NONE** : indique qu'aucun délimiteur ne sera utilisé. (valeur par défaut).

Exemples :

```
open(unit=10, status="old", action="write", &
     position="append")

open(unit=11, file="mon_fichier",           &
     status="replace", form="formatted")
```

13.2 INQUIRE(recl, action, iolength,...)

- **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**.

Exemple :

```
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 (au sens E./S. binaires) d'une structure de type dérivé (sans composante pointeur) faisant partie de la liste spécifiée.

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

Le paramètre **ADVANCE='no'** des instructions **READ/WRITE** (**ADVANCE='yes'** par défaut) permet de ne pas passer à l'enregistrement suivant.

Dans ce cas (lecture avec format explicite et **ADVANCE='no'**) :

- le paramètre **EOR=***nnn* effectue un transfert à l'étiquette **nnn** lorsqu'une fin d'enregistrement est détectée en lecture,
- le paramètre **SIZE=***long* du **READ** permet de récupérer la longueur résiduelle de l'enregistrement physique dans le cas où la fin d'enregistrement est détectée,

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

De même qu'en **Fortran 77** le paramètre **END=nnn** effectue un transfert à l'étiquette **nnn** lorsqu'une fin de fichier est détectée, le paramètre **IOSTAT** permet d'effectuer ce type de détection. Il retourne un entier :

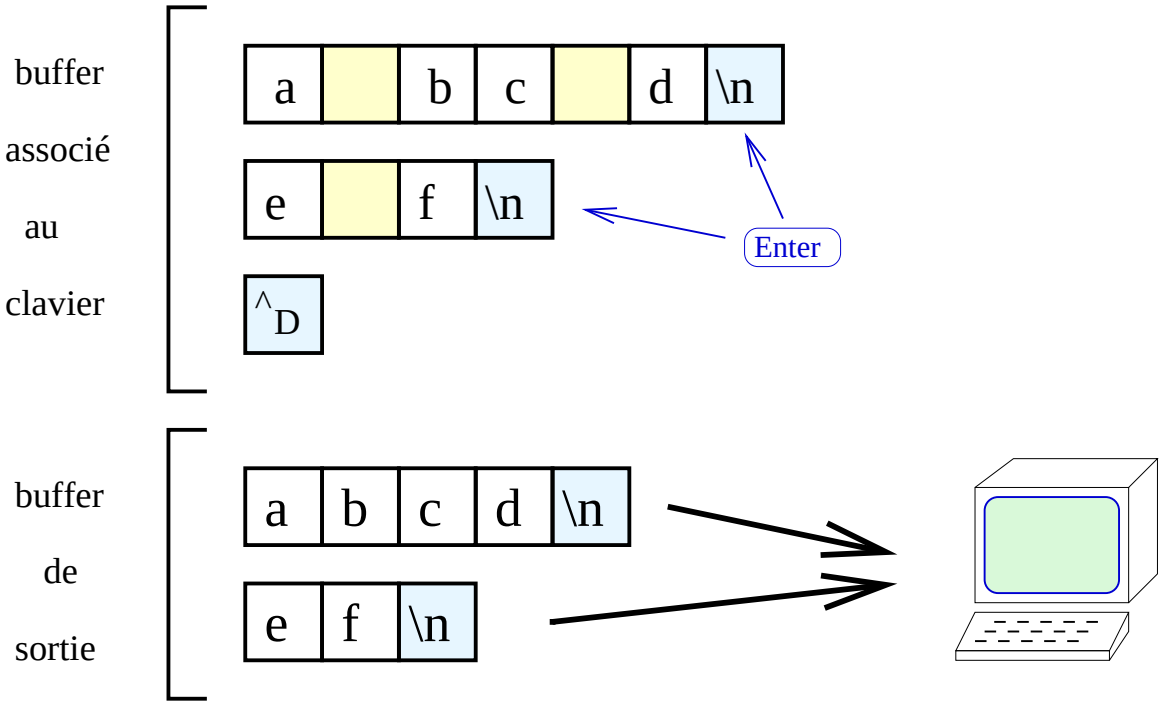
- **positif** en cas d'erreur,
- **négatif** lorsqu'une fin de fichier ou une fin d'enregistrement est atteinte (valeurs dépendant du constructeur),
- **nul** sinon.

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

Nouveautés sur les E/S : entrées-sorties sur les fichiers texte

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.'
  .......
```



13.4 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**. Par exemple :

```
&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 soit assurée via une :*

- option (`-Wf"-P z"` sur Nec)

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

13.5 Spécification de format minimum

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**.

Par exemple :

```
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.

14 Quelques nouvelles fonctions intrinsèques

- ⇒ Conversion entiers/caractères (**`char, ichar,...`**)
- ⇒ Comparaison de chaînes (**`lge, lgt, lle, llt`**)
- ⇒ Manipulation de chaînes (**`adjustl, index, ...`**)
- ⇒ Transformations (**`transfer`**)
- ⇒ Précision/codage numérique (**`tiny, huge, epsilon, nearest, spacing,...`**)
- ⇒ Mesure de temps, date, nombres aléatoires
- ⇒ Opérations sur les bits (**`iand, ior, ishft, ...`**)

14.1 Conversion entiers/caractères (`char, ichar,...`)

- **`CHAR(i,[kind])`**

  $\Longrightarrow$ i^{e}caractère de la table standard (ASCII/EBCDIC) si **`kind`** absent, sinon de la table correspondant à **`kind`** (*constructeur dépendant*).

- **`ACHAR(i)`**

  $\Longrightarrow$ idem **`CHAR`** avec table ASCII.

- **`ICHAR(c)`**

  $\Longrightarrow$ numéro (entier) du caractère **`c`** dans la table à laquelle appartient **`c`** (ASCII/EBCDIC en général).

- **`IACHAR(c)`**

  idem **`ICHAR`** dans la table ASCII.

14.2 Comparaison de chaînes (`lge, lgt, lle, llt`)

- **`LGE(string_a, string_b)`**
  $\Longrightarrow$ *VRAI* si **`string_a`** **après (ou =)** **`string_b`** dans la table ASCII.
  (*Lexically Greater or Equal*)
- **`LGT(string_a, string_b)`**
  $\Longrightarrow$ *VRAI* si **`string_a`** **après** **`string_b`** dans table ASCII.
- **`LLE(string_a, string_b)`**
  $\Longrightarrow$ *VRAI* si **`string_a`** **avant (ou =)** **`string_b`** dans table ASCII.
- **`LLT(string_a, string_b)`**
  $\Longrightarrow$ *VRAI* si **`string_a`** **avant** **`string_b`** dans table ASCII.

Remarques :

- 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 **`/=`**.

14.3 Manipulation de chaînes (`adjustl, index, ...`)

- **`ADJUSTL(string)`**
  $\Longrightarrow$ débarrasse **`string`** de ses blancs de tête (cadrage à gauche) et complète à droite par des blancs.
- **`ADJUSTR(string)`** $\Longrightarrow$ idem **`ADJUSTL`** mais à droite.
- **`INDEX(string, substring [,back])`**
  $\Longrightarrow$ 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)`**
  $\Longrightarrow$ longueur (entier) de la chaîne débarrassée de ses blancs de fin.
- **`SCAN(string, set [,back])`**
  $\Longrightarrow$ 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])`**
  $\Longrightarrow$ 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)`**
  $\Longrightarrow$ chaîne obtenue en concaténant **`ncopies`** copies de **`string`**.
- **`TRIM(string)`** $\Longrightarrow$ débarrasse **`string`** de ses blancs de fin.

14.4 Transformation (`transfer`)

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

$\Longrightarrow$ 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 nombre tel que sa représentation physique mémoire contienne celle de source), sinon un scalaire.
- Si **`size`** présent, retourne un vecteur de taille **`size`**.

Nouvelles fonctions intrinsèques : transformation

Exemples :

`TRANSFER(1082130432 , 1.0)` $\Longrightarrow$ **`4.0`** (sur machine IEEE)

`TRANSFER( (/ 1.,2.,3.,4. /) , (/ (0.,0.) /) )` $\Longrightarrow$
`(/ (1.,2.) , (3.,4.) /)`

```
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(:) )
```

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

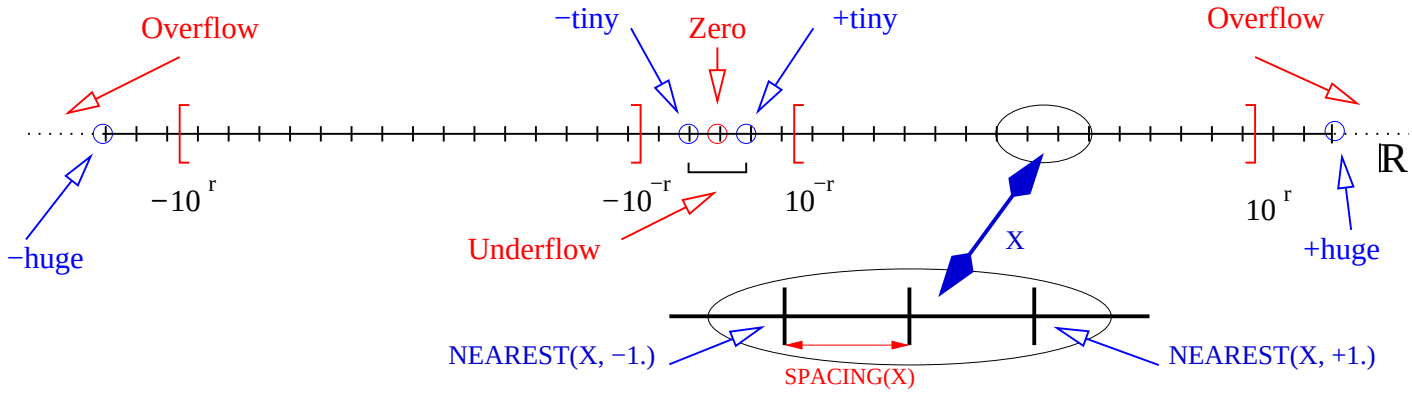
```
integer*8   tampon(4)
character*8 str,ch
real*8      x(3),y(3)
EQUIVALENCE (tampon(1),str) , (tampon(2),x)
ch = str
y(:) = x(:)
```

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))
```

[1] l'**`EQUIVALENCE`** fait partie de la norme Fortran 95 et n'est pas obsolète. Par contre, l'**`EQUIVALENCE`** entre un entier (**`tampon(1)`**) et une variable caractère (**`str`**) est une extension à la norme !

14.5 Précision/codage numérique : `tiny/huge, sign, nearest, spacing, ...`



`TINY(x)` plus petite valeur réelle représentable dans le sous-type de **`x`** (limite d'*underflow*).

`HUGE(x)` plus grande valeur réelle ou entière représentable dans le sous-type de **`x`** (limite d'*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)` $\Longrightarrow$ **`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`**).

Nouvelles fonctions intrinsèques : précision et codage numérique

`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 2^{e} 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 SX5 et IBM SP4.*

14.6 Mesure de temps, date, nombres aléatoires

`CPU_TIME(time)` (**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-progr. 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-progr. 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...

Nouvelles fonctions intrinsèques : mesure de temps, nombres aléatoires

Exemple 1 : génération de deux séries de nombres aléatoires dans un tableau **tab** :

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

Exemple 2 : 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** :

```
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**.

Exemple 3 : 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.'
. . .
```

Nouvelles fonctions intrinsèques : mesure de temps, nombres aléatoires

Exemple 4 : 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 '(47a)', ("-", n=1,47)
  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 '(47a)', ("-", n=1,47)
  !
end program date
```

Voici la sortie correspondante :

```
 -----------------------------------------------
| Test date_and_time ==> 15/03/2005 - 15H15M44S |
 -----------------------------------------------
```

14.7 Opérations sur les bits (`iand, ior, ishft, ...`)

`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.

Nouvelles fonctions intrinsèques : opérations sur les bits

`IBCLR(i,pos)` fonction retournant un entier identique à **`i`** avec le **`pos`**ième bit mis à zéro.

`IBSET(i,pos)` fonction retournant un entier identique à **`i`** avec le **`pos`**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`**).

Remarque : ces fonctions ont été étendues pour s'appliquer aussi à des tableaux d'entiers.

Norme 95 : le sous-programme **`MVBITS`** est "**`pure`**" et "**`elemental`**".

A Annexe : paramètre KIND et précision des nombres

⇒ Sur IBM/SP4

⇒ Sur NEC/SX5

Annexe A – Paramètre KIND : précision des nombres sur IBM/SP4

A.1 Sur IBM/SP4

- **Entiers**

  kind = **1** $\Longrightarrow$ 1 **octet** : $-128 \leq i \leq 127$

  kind = **2** $\Longrightarrow$ 2 **octets** : $-2^{15} \leq i \leq 2^{15}-1$

  kind = **4** $\Longrightarrow$ 4 **octets** : $-2^{31} \leq i \leq 2^{31}-1$

  kind = **8** $\Longrightarrow$ 8 **octets** : $-2^{63} \leq i \leq 2^{63}-1$

- **Réels**

  kind = **4** $\Longrightarrow$ 4 **octets** : $1.2 \times 10^{-38} \leq \mid r \mid \leq 3.4 \times 10^{38}$

  6 chiffres significatifs décimaux.

  kind = **8** $\Longrightarrow$ 8 **octets** : $2.2 \times 10^{-308} \leq \mid r \mid \leq 1.8 \times 10^{308}$

  15 chiffres significatifs décimaux.

  kind = **16** $\Longrightarrow$ 16 **octets** : $2.2 \times 10^{-308} \leq \mid r \mid \leq 1.8 \times 10^{308}$

  31 chiffres significatifs décimaux.

- **Complexes**

  kind = **4** $\Longrightarrow$ (4,4) **octets** $\equiv$ complex*8 (f77)

  kind = **8** $\Longrightarrow$ (8,8) **octets** $\equiv$ complex*16 (f77)

  kind = **16** $\Longrightarrow$ (16,16) **octets** $\equiv$ complex*32 (f77)

- **Logiques**

  kind = **1** $\Longrightarrow$ 1 **octet** : 01 = .true. et 00 =.false.

  kind = **2** $\Longrightarrow$ 2 **octets** : 0001 = .true. et 0000 =.false.

  kind = **4** $\Longrightarrow$ 4 **octets** : 0..1 = .true. et 0..0 =.false.

  kind = **8** $\Longrightarrow$ 8 **octets** : 0...1 = .true. et 0...0 =.false.

- **Caractères** : kind = **1** $\Longrightarrow$ jeu **ASCII**

A.2 Sur NEC/SX5

Types et sous-types disponibles avec les options **-dW** et **-dw**.

- **Entiers**

  kind = **2** $\Longrightarrow$ 2 **octets** : $-2^{15} \leq i \leq 2^{15}-1$

  kind = **4** $\Longrightarrow$ 4 **octets** : $-2^{31} \leq i \leq 2^{31}-1$

  kind = **8** $\Longrightarrow$ 8 **octets** : $-2^{63} \leq i \leq 2^{63}-1$

- **Réels**

  kind = **4** $\Longrightarrow$ 4 **octets** : $1.2 \times 10^{-38} \leq \mid r \mid \leq 3.4 \times 10^{38}$

  6 chiffres significatifs décimaux.

  kind = **8** $\Longrightarrow$ 8 **octets** : $2.2 \times 10^{-308} \leq \mid r \mid \leq 1.8 \times 10^{308}$

  15 chiffres significatifs décimaux.

  kind = **16** $\Longrightarrow$ 16 **octets** : $2.2 \times 10^{-308} \leq \mid r \mid \leq 1.8 \times 10^{308}$

  31 chiffres significatifs décimaux.

- **Complexes**

  kind = **4** $\Longrightarrow$ (4,4) **octets** $\equiv$ complex*8 (f77)

  kind = **8** $\Longrightarrow$ (8,8) **octets** $\equiv$ complex*16 (f77)

  kind = **16** $\Longrightarrow$ (16,16) **octets** $\equiv$ complex*32 (f77)

- **Logiques**

  kind = **1** $\Longrightarrow$ 1 **octet** : 01 = .true. et 00 =.false.

  kind = **4** $\Longrightarrow$ 4 **octets** : 0..1 = .true. et 0..0 =.false.

  kind = **8** $\Longrightarrow$ 8 **octets** : 0...1 = .true. et 0...0 =.false.

- **Caractères** : kind = **1** $\Longrightarrow$ jeu **ASCII**

Annexe A – Paramètre KIND : précision des nombres sur NEC/SX5

Page réservée pour vos notes personnelles...

B Annexe : exercices

⇒ Exercices : énoncés
⇒ Exercices : corrigés

B.1 Exercices : énoncés

Exercice 1 :

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 2 :

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, ...

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 afi n de vérifi er son contenu.

Exercice 2_suppl :

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 vérifi er que la trace de la matrice est inchangée.

Exercice 3 :

Compiler et exécuter le programme contenu dans les fi chiers **`exo3.f90`**, **`mod1_exo3.f90`** et **`mod2_exo3.f90`** :

```
program exo3
      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 exo3

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

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
```

Recommencez en plaçant les modules dans un répertoire différent de celui où se trouve le programme principal.

Exercice 4 :

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 fi chier (cf. fi chier **`palindrome`**) contenant des phrases (suite de mots séparés par des blancs qu'il faudra supprimer – phase de *compression*).

Exercice 5 :

Compléter le programme contenu dans le fi chier **`exo5.f90`** jusqu'à ce qu'il s'exécute correctement : les 2 matrices imprimées devront être identiques.

```
program exo5
  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) /), (/ n,m /))
  print *,"Matrice a avant appel à sp :"
  print *,"--------------------------"
  do i=0,size(a,1)-1
    print *,a(i,:)
  enddo
  call sp(a)
end program exo5
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 6 :

Écrire un programme permettant l'impression des **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 7 :

Cet exercice reprend le module **`matrix`** des chapitres 10 et 11 (cf. page 179) du support de cours. Il est stocké (avec un programme principal d'utilisation complet) dans le fi chier **`exo7.f90`**.

Complétez le module **`matrix`** en défi nissant un opérateur **`.vp.`** permettant le calcul des valeurs propres d'un objet de type **`OBJ_MAT`**. Utilisez par exemple le sous-programme **`EVLRG`** (calculant les valeurs propres d'une matrice réelle d'ordre **`N`**) de la bibliothèque IMSL dont la séquence d'appel est :

```
CALL EVLRG(N, A, LDA, EVAL)   avec :

     - N    : nombre de lignes de A              (entrée)
     - A    : matrice réelle d'ordre N           (entrée)
     - LDA  : "Leading Dimension of A" (N ici)   (entrée)
     - EVAL : vecteur contenant les N valeurs    (sortie)
              propres complexes
```

En entrée :

le programme principal lit (en "format libre" et avec le paramètre **`ADVANCE="NO"`**) un fi chier **`exo7.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 fi chier (avec une matrice d'ordre **`N=4`**) est contenu dans **`exo7.data`**.Vous devez alors trouver les valeurs propres suivantes : **`(4.,0.), (3.,0.), (2.,0.), (1.,0.)`**.

Notes :

- deux *méthodes* ont déjà été ajoutées au module **`matrix`** :
  - la fonction logique **`erreur()`** permettant de tester la bonne convergence du calcul effectué par **`EVLRG`**,
  - le sous-programme **`imp_vp(vect_complexe)`** pour faciliter l'impression des valeurs propres.
- pour l'accès à la bibliothèque IMSL (version 77) de l'IDRIS, lisez au préalable la *"news IMSL"* sur les machines de calcul.
  À défaut, consultez le **README** du répertoire **lapack**.

Exercice 8 :

Soit le programme principal contenu dans le fi chier **`exo8.f90`** (ou **`exo8.f`**) :

```
program exo8
 use music
 type(musicien) :: mus_mort_le_plus_jeune
 call init
 call tri(critere="nom")
 call tri(critere="annee")
 mus_mort_le_plus_jeune = tab_mus
 print *
 print *,"Le musicien mort le plus jeune est : ", &
         nom(mus_mort_le_plus_jeune)
end program exo8
```

Dans le module **`music`** à créer, défi nir :

- le type **`musicien`** et un tableau **`tab_mus`** de ce type (dimensionné à 30).
- le sous-programme **`init`** devant lire le contenu du fi chier **`musiciens`** (ce fi chier contient une liste de compositeurs avec leurs années de naissance et de mort : éditez-le au préalable afi n de connaître son formatage) afi n de valoriser le tableau **`tab_mus`** et l'imprimer,
- le sous-programme **`tri`** qui trie et imprime 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 9 :

Même exercice que précédemment mais avec utilisation d'une *liste chaînée* simple ou double en partant du programme principal suivant contenu dans le fi chier **`exo9.f90`** (ou **`exo9.f`**) :

```
program exo9
  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 exo9
```

Remarque : **`debut`** correspond au pointeur de début de liste.

B.2 Exercices : corrigés

Exercice 1 : corrigé

```
program exo1
  implicit none
  integer, parameter      :: n = 10
  real   , dimension(n)   :: diag = 1.
  real   , dimension(n,n) :: mat_ident
  logical, dimension(n,n) :: mask
  real, dimension(n*n)    :: vect = 0.
  character(len=8)        :: fmt = "(00f3.0)"
  integer                 :: i, j
  !
  write(fmt(2:3), '(i2)')n  ! Format d'impression
!
!==============> Première solution :
  mask = reshape((/ ((i == j, i=1,n), j=1,n) /), &
                 shape = (/ n,n /))
  mat_ident = unpack(diag, mask, 0.)
!
!==============> Deuxième solution :
  vect(1:n*n:n+1) = 1.
  mat_ident = reshape(vect, shape = (/ n,n /))
!
!==============> Troisième solution :
  mat_ident(:,:) = 0.
  mat_ident(:,1) = 1.
  mat_ident(:,:) = cshift(array=mat_ident, &
                          shift=(/ (-i,i=0,n-1) /), &
                          dim=2)
  !
  do i=1,n   ! Impression matrice identité
    print fmt,mat_ident(i,:)
  end do
end program exo1
```

Annexe B2 – Exercice 2 : corrigé

Exercice 2 : corrigé

```
program exo2
  implicit none
  !
  ! On décide que les entiers err,n,m,i sont < 99
  !
  integer, parameter      :: p = selected_int_kind(2)
  integer(kind=p)         :: n,m
  integer(kind=p)         :: err,i
  integer, dimension(:,:),allocatable :: mat
  !
  ! Lecture des dimensions de la matrice
  !
  print *,"Nombre de lignes? :"  ; read(*,*)n
  print *,"Nombre de colonnes? :"; read(*,*)m
  !
  ! Allocation de la matrice
  !
  allocate(mat(n,m),stat=err)
  if (err /= 0) then
    print *,"Erreur d'allocation"; stop 4
  endif
  !
  ! Remplissage des lignes paires avec l'entier 1
  !
  mat(2:n:2,:) = 1
  !
  ! Remplissage lignes impaires avec les entiers 1,2,...
  !
  mat(1:n:2,:)=reshape((/ (i,i=1,size(mat(1:n:2,:))) /),&
               shape=shape(mat(1:n:2,:)),order=(/ 2,1 /))
  !
  ! On imprime la matrice obtenue après remplissage
  !
  do i=1,n
    print *,mat(i,:)
  enddo
end program exo2
```

Exercice 2_suppl : corrigé

```
program exo2_suppl
  implicit none
  real, dimension(:,:), allocatable :: A
  integer                           :: N, i, j
  real                              :: trace
  !------- Allocation/initialisation de A --------------
  print *, "N (multiple de 4 < 32) ?" ; read *, N
  allocate(A(N,N))
  A=reshape(source = (/ (real(i), i=1,N*N) /),     &
            shape  = (/ N,N /), order  = (/ 2,1 /))
  print "(/,A,/)", "Matrice à transformer :"
  do i=1,N
    print "(16F5.0)", A(i,:)
  end do
  trace = sum(pack(A, mask=reshape(           &
            source=(/ ((i==j,i=1,N),j=1,N) /),&
            shape =shape(A))))
  print *, "trace=", trace
  !------------- 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
  print "(/,A,/)", "Matrice transformée :"
  do i=1,N
    print "(16F5.0)", A(i,:)
  end do
  trace = sum(pack(A,mask=reshape(           &
            source=(/ ((i==j,i=1,N),j=1,N) /),&
            shape =shape(A))))
  print *, "trace=", trace
end program exo2_suppl
```

Annexe B2 – Exercice 3 : corrigé

Exercice 3 : corrigé

Sur NEC/SX5 (Uqbar) et sa frontale SGI (Rhodes)

```
Frontale-Rhodes> sxf90 -c mod1_exo3.f90
Frontale-Rhodes> sxf90 -c mod2_exo3.f90
Frontale-Rhodes> sxf90 exo3.f90 mod1_exo3.o mod2_exo3.o
                       -o $HOMESX5/exo3
  NEC/SX5 Uqbar> exo3
```

Remarque : si les modules sont situés dans un répertoire **`rep1`** différent de celui (**`rep2`**) d'exo3, utiliser l'option **`-I`** :

```
cd $HOME/rep1
Frontale-Rhodes> sxf90 -c mod1_exo3.f90 mod2_exo3.f90
Frontale-Rhodes> cd ../rep2
Frontale-Rhodes> sxf90 exo3.f90 -I ../rep1 ../rep1/mod*.o
                       -o $HOMESX5/exo3
  NEC/SX5 Uqbar> exo3
```

Sur IBM/SP4

```
IBM/SP4> f90 -c mod1_exo3.f90 mod2_exo3.f90
IBM/SP4> f90 exo3.f90 mod1_exo3.o mod2_exo3.o -o exo3
IBM/SP4> exo3
```

Exemple de makefile sur IBM/SP4 :

```
OBJSEXO3    = mod1_exo3.o mod2_exo3.o exo3.o
FC          = f90
FLAGS       = -qsource -O2
.SUFFIXES : .f90
all:    exo3
.f90.o:
        $(FC) $(FLAGS) -c $<
mod2_exo3.o : mod1_exo3.o
              $(FC) $(FLAGS) -c $<
exo3.o: mod2_exo3.o
        $(FC) $(FLAGS) -c $<
exo3:   $(OBJSEXO3)
        $(FC) -o $@ $(OBJSEXO3)
        $@
```

Exercice 4 : corrigé

```
program exo4
  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
    !
    ! Récup. longueur chaîne entrée (sans blancs de fin).
    !
    long_util = len_trim( chaine )

    if( palind( chaine(:long_util) ) ) then
      print *, chaine(:long_util)," est un palindrome"
    else
      print *, chaine(:long_util)," n'est pas un palindrome"
    endif
  enddo
  if ( choix == 2 ) close( unit=1 )
```

```
contains

  function palind( chaine )
    logical                 :: palind    !<-- Retour fonction
    character(len=*)        :: chaine    !<-- Arg. muet
    character(len=1), &
      dimension(len(chaine)):: tab_car   !<-- Tabl. automatique
    integer                 :: long_util !<-- Var. locale
    !
    ! 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     !<-- Tabl. automatique

    m(:) = tab_car(:) /= ' '
    long_util = count( mask=m(:) )
    tab_car(:long_util) = pack(array=tab_car(:), mask=m(:))
  end function compression

end program exo4
```

Annexe B2 – Exercice 5 : corrigé

```
program exo5
! Du fait de la déclaration de l'argument a de sp
! avec un profil implicite, l'interface doit être
! explicite,  d'où l'ajout du "bloc interface".
! Dans ce contexte, seul le profil de a est passé à sp :
! les bornes inférieures nulles ne le sont pas ! À moins
! d'une déclaration explicite "dimension(0:,0:)" dans sp,
! les bornes inférieures sont égales à 1 par défaut.
!
  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) /), (/ n,m /))
  print *,"Matrice a avant appel à sp :"
  print *,"--------------------------"
  print *
  do i=0,size(a,1)-1
    print *,a(i,:)
  enddo
  call sp(a)
end program exo5
!-----------------
subroutine sp(a)
  integer, dimension(:,:) :: a
  integer                 :: i
  print *
  print *,"Matrice a dans sp :"
  print *,"-----------------"
  print *
  do i=1,size(a,1)
    print *,a(i,:)
  enddo
end subroutine sp
```

Annexe B2 – Exercice 6 : corrigé

```
program exo6
!-----------
implicit none
type ligne
  integer, dimension(:), pointer :: p
end type ligne
type(ligne), dimension(:), allocatable :: triangle
integer       :: i, n, err
character(11) :: fmt = "(00(i5,1x))"
do
  write(6,advance='no',fmt="('Ordre du triangle ? :')")
  read(5, *)n
  if (n >= 20) exit       ! On limite à 19 lignes
  write(fmt(2:3), '(i2)')n! Construction du format
  !                       ! de sortie
  !--- On alloue le nombre de lignes du triangle.
  allocate(triangle(n), stat=err)
  if (err /= 0) stop "Erreur à l'allocation de triangle"
  do i=1,n
    !--- Pour chaque ligne du triangle, allocat. du nombre
    !--- de colonnes.
    allocate(triangle(i)%p(i), stat=err)
    if (err /= 0) stop "Erreur à l'allocation d'une &
                       &ligne de triangle"
    !
    !-Valorisation éléments extrêmes de la ligne courante
    !-puis les autres éléments à partir de la 3ème ligne.
    triangle(i)%p((/ 1,i /)) = 1
    if (i > 2)  &
      triangle(i)%p(2:i-1)   = triangle(i-1)%p(2:i-1) + &
                               triangle(i-1)%p(1:i-2)
    print fmt,triangle(i)%p     ! Impression de la ligne
  end do
  !
  !-- Une fois le triangle construit et imprimé, on libère
  !-- chaque ligne et le tableau triangle.
  do i=1,n
    deallocate(triangle(i)%p)
  end do
  deallocate(triangle)
end do
end program exo6
```

Exercice 7 : corrigé

```
module matrix
  integer           :: nb_lignes, nb_col
  integer, private  :: err

  type OBJ_MAT
    private
    logical                          :: CreeParFonction
    integer                          :: n=0, m=0
    real, dimension(:,:), pointer :: ptr_mat => NULL()
  end type OBJ_MAT

  private  :: add, trans, taille_mat, valorisation, affect
  private  :: ramasse_miettes, 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
!-------
```

Annexe B2 – Exercice 7 : corrigé

```
subroutine valorisation(a,t)
  type(OBJ_MAT),      intent(inout) :: a
  real, dimension(:), intent(in)    :: t
   ....
end subroutine valorisation
!--------------------------------
subroutine ramasse_miettes(a)
  type(OBJ_MAT), intent(in) :: a
  type(OBJ_MAT)             :: temp

  temp%ptr_mat => a%ptr_mat
  call poubelle(temp)
end subroutine ramasse_miettes
!--------------------------------
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(a%n)   :: val_propres
  if (associated(a%ptr_mat)) then
   call evlrg(a%n, a%ptr_mat, a%n, val_propres)
  else
   Stop "Objet non existant"
  end if
end function val_propres
```

```
  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 *,a%ptr_mat(i,:)
    enddo
    if (a%CreeParFonction) call ramasse_miettes(a)
  end subroutine imp
  !-------------------------------
  logical function erreur()
    erreur = iercd() /= 0
  end function erreur
  !-------------------------------
  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
```

Annexe B2 – Exercice 7 : corrigé

```
program exo7
  use matrix
  type(OBJ_MAT)   :: u
  real,    dimension(:), allocatable :: val_init
  complex, dimension(:), allocatable :: val_pr
  open(unit=10, file="exo7.data", form='formatted', &
       action="read")
  read(10, advance='no', fmt='(i1)') nb_lignes
  nb_col = nb_lignes
  allocate(val_init(nb_lignes*nb_col))
  allocate(val_pr(nb_col))
  read(10, *) val_init
  close(10)
  u = val_init
  deallocate(val_init)
  call imp(u)
  !---------------
   val_pr = .vp. u
  !---------------
  if (erreur()) then
    print *,"La méthode diverge"
  else
    call imp_vp(val_pr)
  end if
  deallocate(val_pr)
  call poubelle(u)
end program exo7
```

Exercice 8 : corrigé

```
module music
  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 assignment(=)
    module procedure mort_le_plus_jeune
  end interface
```

Annexe B2 – Exercice 8 : corrigé

```
contains
  !--------------------------------
  subroutine init
    integer   :: eof,i,err
    !
    ! Valorisation du tableau de musiciens
    !
    open(1,file="musiciens",action="read",status="old")
    nb_mus = 0
    do
      read(1,'(a16,1x,a21,2(1x,i4))', iostat=eof) &
                                  tab_mus(nb_mus+1)
      if (eof /= 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),stat=err)
    if (err /= 0) then
      print *,"Erreur d'allocation"
      stop 4
    endif
    !
    ! 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,a16,1x,a21,2(1x,i4)))') &
         (tab_mus(i),i=1,nb_mus)
  end subroutine init
```

Annexe B2 – Exercice 8 : corrigé

```
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.
  !
  character(len=*)   :: critere
  logical            :: expr, tri_termine
  character(len=13)  :: mode
  !
  do
    tri_termine = .true.
    do i=1,nb_mus-1
      select case(critere)
        case("nom")
          mode = "alphabétique"
          expr = tab_ptr_musicien(i)%ptr%nom > &
                 tab_ptr_musicien(i+1)%ptr%nom)
        case("annee")
          mode = "chronologique"
          expr = tab_ptr_musicien(i)%ptr%annee_naiss > &
                 tab_ptr_musicien(i+1)%ptr%annee_naiss
        case default
      end select
      if (expr) then
        !--Permutation des deux associations----------
        tab_ptr_musicien(i:i+1)   =  &
        tab_ptr_musicien(i+1:i:-1)
        !---------------------------------------------
        tri_termine = .false.
      endif
    enddo
    if (tri_termine) exit
  enddo
  !
  print '(/, a, a, a, /)', '---- Liste ', mode,
        ' des musiciens ----'
  write(*,'((5x,a16,1x,a21,2(1x,i4)))') &
       (tab_ptr_musicien(i)%ptr,i=1,nb_mus)
  end subroutine tri
```

Annexe B2 – Exercice 8 : corrigé

```
  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
  !----------------------
  subroutine mort_le_plus_jeune(mus,tab_mus)
    !-- Surcharge de l'opérat. d'affectation "=".
    !
    type(musicien), intent(out) :: mus
    type(musicien), dimension(:), intent(in) :: tab_mus
    mus = tab_mus(1)
    do i=2,nb_mus
      !
      ! Ici l'utilisation de l'opérateur < provoque
      ! l'appel à la fonction "longevite". En fait :
      ! tab_mus(i) < mus  <=> longevite(tab_mus(i),mus)
      !
      if (tab_mus(i) < mus) then
        mus = tab_mus(i)
      endif
    enddo
  end subroutine 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
    !  write(nom, '(a16,1x,a21)') mus%prenom,mus%nom
    nom = trim(mus%prenom)//' '//mus%nom
  end function nom

end module music
```

Exercice 9 : solution avec liste chaînée simple

```
module music
  !---------------------------------
  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
```

Annexe B2 – Exercice 9 : corrigés

```
contains
  !---------------------------------
  subroutine init
    type(musicien)          :: mus
    type(musicien), pointer :: ptr_precedent, ptr_courant
    integer                 :: eof, err

    nullify(debut)
    nullify(mus%ptr)
    open(1,file="musiciens",action="read",status="old")
    do
      read(1,'(a16,1x,a21,2(1x,i4))',iostat=eof) &
                                mus%prenom,      &
                                mus%nom,         &
                                mus%annee_naiss, &
                                mus%annee_mort
      if (eof /= 0) exit
      allocate(ptr_courant, stat=err)
      if (err /= 0) stop 4
      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
```

```
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*(*),  intent(in) :: critere
  type(musicien), pointer    :: ptr_courant, &
                                ptr_precedent, temp
  logical                    :: tri_termine, expr
  character(len=13)          :: mode

  do
    tri_termine = .true.
    ptr_courant   => debut
    ptr_precedent => debut
    do
      if (.not.associated(ptr_courant%ptr)) exit
      select case(critere)
        case("nom")
          mode = "alphabétique"
          expr = ptr_courant%nom > ptr_courant%ptr%nom
        case("annee")
          mode = "chronologique"
          expr = ptr_courant%annee_naiss > &
                 ptr_courant%ptr%annee_naiss
        case default
      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 ',mode,' des musiciens ----'
  print *
  call liste
end subroutine tri
!--------------------------------
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".
    ! En fait : 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
  ! write(nom, '(a16,1x,a21)')mus%prenom,mus%nom
  nom = trim(mus%prenom)//' '//mus%nom
end function nom
```

```
  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,a16,1x,a21,2(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
```

Exercice 9 : solution avec liste chaînée double

```
module music
  !---------------------------------
  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
  !---------------------------------
contains
```

Annexe B2 – Exercice 9 : corrigés

```
subroutine init
  type(musicien)           :: mus
  type(musicien), pointer :: ptr_precedent, ptr_courant
  integer                  :: eof, err

  nullify(debut)
  nullify(mus%ptr_precedent)
  nullify(mus%ptr_suivant)
  open(1,file="musiciens",action="read",status="old")
  do
    read(1,'(a16,1x,a21,2(1x,i4))',iostat=eof)   &
                                 mus%prenom,     &
                                 mus%nom,        &
                                 mus%annee_naiss, &
                                 mus%annee_mort
    if (eof /= 0) exit
    allocate(ptr_courant, stat=err)
    if (err /= 0) stop 4
    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
```

```
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*(*),  intent(in) :: critere
  type(musicien), pointer    :: ptr_courant, ptr
  integer                    :: err
  logical                    :: tri_termine, expr
  character(len=13)          :: mode

  do
    tri_termine = .true.
    ptr_courant   => debut
    do
      if(.not.associated(ptr_courant%ptr_suivant))exit
      select case(critere)
        case("nom")
          mode = "alphabétique"
          expr = ptr_courant%nom > &
                 ptr_courant%ptr_suivant%nom
        case("annee")
          mode = "chronologique"
          expr = ptr_courant%annee_naiss > &
                 ptr_courant%ptr_suivant%annee_naiss
        case default
      end select
```

Annexe B2 – Exercice 9 : corrigés

```
        if (expr) then
          allocate(ptr, stat=err)
          if (err /= 0) stop 4
          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 ',mode,' des musiciens ----'
    print *
    call liste
  end subroutine tri
  !
  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
!--------------------------------
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".
    ! En fait : 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
    ! passe en argument.
    !
    type(musicien), intent(in) :: mus
    character(len=38)          :: nom
    nom = trim(mus%prenom)//' '//mus%nom
  end function nom
  !--------------------------------
  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,a16,1x,a21,2(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,a16,1x,a21,2(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
```

C Annexe : apports de la norme 95

⇒ Procédures “pure”

⇒ Procédures “elemental”

⇒ Le “bloc FORALL”

Note :

les autres apports de la norme 95 ont été intégrés dans les divers chapitres concernés de ce manuel

(cf. chap. 1.5 page 16)

C.1 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**.

Voici un 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 ou **`pointer`** 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).

C.2 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 **in**/**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.

C.3 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.

Exemple 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.

Exemple 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
```

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)
```

Exemple 3 : 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**.

```
program exemple3
 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,:))
  ENDWHERE
  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 exemple3
```

Index

– K –

– L –

– M –

– N –

– O –

– P –

– R –

– S –

– T –

– U –

– V –

– W –

– Z –