# Cours Fortran 95

Patrick Corde et Hervé Delouis

26 mai 2000

Version 6.1





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. Pour une étude exhaustive, consulter les manuels référencés aux paragraphes 1.9 et 1.10. Une version HTML de ce support, avec des notes de cours repérées par le signe `<*>`, est disponible sur le serveur Web de l'IDRIS. Cf. `http://www.idris.fr/data/cours/lang/f90/F90_cours.html`

Patrick Corde `<corde@idris.fr>`
Hervé Delouis `<delouis@idris.fr>`

© Institut du Développement et des Ressources en Informatique Scientifique (C.N.R.S.)
Boîte postale 167 – 91403 ORSAY CEDEX

26 mai 2000

Reproduction totale de ce document interdite sans autorisation des auteurs.
Reproduction partielle autorisée pour l'usage du copiste.

I001 260p





## Table des matières

**1 Introduction** **9**
1.1 Historique . . . 10
1.2 Compatibilité norme 77/90 . . . 12
1.3 Apports de Fortran 90 . . . 13
1.4 Aspects obsolètes . . . 14
1.5 Évolution : nouveautés de la norme Fortran 95 . . . 16
1.6 Fortran 90 sur IBM-RS/6000 . . . 18
1.7 Fortran 90 sur NEC SX-5 . . . 19
1.8 Bibliographie . . . 20
1.9 Documentation . . . 22

**2 Généralités** **25**
2.1 Structure d'un programme . . . 26
2.2 Éléments syntaxiques . . . 27
2.2.1 Les identificateurs . . . 27
2.2.2 Le format libre . . . 28
2.2.3 Les commentaires . . . 29
2.2.4 Le format fixe . . . 31
2.2.5 Les déclarations . . . 32
2.2.6 Typage et précision des nombres : paramètre `KIND` 36
2.3 Compilation, édition des liens, exécution . . . 43





**3 Types dérivés** **45**
3.1 Définition et déclaration de structures . . . 46
3.2 Initialisation (constructeur de structure) . . . 47
3.3 Symbole % d'accès à un champ . . . 48
3.4 Types dérivés et procédures . . . 49
3.5 Conclusion et rappels . . . 52

**4 Programmation structurée** **57**
4.1 Introduction . . . 58
4.2 Boucles `DO` . . . 59
4.3 Construction `SELECT-CASE` . . . 65

**5 Extensions tableaux** **67**
5.1 Définitions (rang, profil, étendue,...) . . . 68
5.2 Manipulations de tableaux (conformance, section, taille, constructeur,...) . . . 70
5.3 Tableau en argument d'une procédure (taille et profil implicites) . . . 77
5.4 Section de tableau non contigué en argument d'une procédure . . . 80
5.5 Fonctions intrinsèques tableaux . . . 82
5.5.1 Interrogation (`maxloc, lbound, shape,..` ) . . . 83
5.5.2 Réduction (`all, any, count, product,...`) . . . 81
5.5.3 Multiplication (`matmul, dot_product,...`) . . . 90

5.5.4 Construction et transformation (`reshape, cshift, pack, merge, transpose,...`) . . . 92
5.6 Instruction et bloc `WHERE` . . . 102

**6 Gestion mémoire 105**
6.1 Tableaux automatiques . . . 106
6.2 Tableaux dynamiques (`allocatable`, profil différé) . . . 107

**7 Pointeurs 109**
7.1 Définition, états d'un pointeur . . . 110
7.2 Déclaration d'un pointeur . . . 111
7.3 Symbole => . . . 112
7.4 Symbole = appliqué aux pointeurs . . . 115
7.5 Allocation dynamique de mémoire . . . 117
7.6 Instruction `NULLIFY` . . . 118
7.7 Fonction intrinsèque `ASSOCIATED` . . . 119
7.8 Situations à éviter . . . 120
7.9 Déclaration de "tableaux de pointeurs" . . . 122
7.10 Passage en argument de procédures . . . 124

**8 Interface de procédures et modules 125**
8.1 Interface "implicite" : définition . . . 126
8.3 Arguments : attributs `INTENT` et `OPTIONAL` . . . 128
8.4 Passage d'arguments par mot-clé . . . 129





8.5 Interface "explicite" : procédure interne (`CONTAINS`) . . . 130
8.6 Interface "explicite" : 5 cas possibles . . . 131
8.7 Interface "explicite" : bloc interface . . . 132
8.8 Interface "explicite" : ses apports . . . 135
8.9 Interface "explicite" : module avec bloc interface (`USE`) . . . 136
8.10 Interface "explicite" : module avec procédure . . . 138
8.11 Cas d'interface "explicite" obligatoire . . . 139

**9 Interface générique 141**
9.1 Introduction . . . 142
9.2 Exemple . . . 143
9.3 Exemple avec `module procedure` . . . 147

**10 Surcharge d'opérateurs 151**
10.1 Introduction . . . 152
10.2 Interface `operator` . . . 154
10.3 Interface `assignment` . . . 156

**11 Contrôle de visibilité 159**
11.1 Introduction . . . 160
11.2 Instruction `PRIVATE` et `PUBLIC` . . . 161
11.3 Attribut `PRIVATE` et `PUBLIC` . . . 162
11.4 Type dérivé semi-privé . . . 163
11.5 Paramètre `ONLY` de l'instruction `USE` . . . 170





**12 Procédures récursives 173**
12.1 Clause `RESULT` . . . 174
12.2 Exemple . . . 175

**13 Nouveautés sur les E/S 177**
13.1 `OPEN (status, position, action, delim,...)` . . . 178
13.2 `INQUIRE (recl, action, iolength, delim,...)` . . . 180
13.3 Entrées-sorties sur les fichiers texte (`advance='no'`) . . . 181
13.4 Instruction `NAMELIST` . . . 183
13.5 Spécification de format minimum . . . 184

**14 Quelques nouvelles fonctions intrinsèques 185**
14.1 Conversions entiers/caractères (`char, ichar,...`) . . . 186
14.2 Comparaison de chaînes (`lge, lgt, lle, llt`) . . . 187
14.3 Manipulations de chaînes (`adjustl, index, scan,...`) 188
14.4 Transformations (`transfer`) . . . 189
14.5 Précision et codage numérique (`tiny, huge,...`) . . . 190
14.6 Mesure de temps (`date_and_time, system_clock,...`) 191
14.7 Opérations sur les bits (`iand, ieor, ior, ishft,...`) 192

**A Annexe : paramètre KIND et précision des nombres**
A.1 Sur IBM RS/6000 . . . 196
A.2 Sur NEC SX5 . . . 197

**B Annexe : exercices 199**





B.1 Exercices : énoncés . . . 200
B.2 Exercices corrigés . . . 207

**C Annexe : zones mémoires *anonymes dormantes* 235**
C.1 A propos de l'exemple du chapitre 11 . . . 235

**D Annexe : exemples complémentaires 241**
D.1 Fonction `CSHIFT` : dérivée d'une matrice . . . 241
D.2 Fonctions `PACK` et `UNPACK` et matrice tridiagonale . . . 242

**E Annexe : expressions d'initialisations 243**

**F Annexe : pointeurs : listes chaînées, `COMMON` 245**

**G Annexe : contrôle de procédure F77 via interface "générique et explicite" 247**

**H Annexe : apports de la norme 95 249**

**Index 257**

# Cours Fortran 95
# 1 – Introduction : plan



## 1 Introduction


**1.1 Historique**

**1.2 Compatibilité norme 77/90**

**1.3 Apports de Fortran 90**

**1.4 Aspects obsolètes**

**1.5 Évolution : nouveautés Fortran 95**

**1.6 Fortran 90 sur IBM-RS/6000**

**1.7 Fortran 90 sur NEC SX-5**

**1.8 Bibliographie**

**1.9 Documentation**






# 1.1 – Introduction : 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.



# 1.1 – 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.
- 1999 – sur Cray T3E puis IBM RS/6000 $\Longrightarrow$ **Fortran 95**





# 1.2 – Introduction : compatibilité

- La norme 77 est totalement incluse dans la norme 90.
- Quelques comportements différents :
  - beaucoup plus de fonctions/sous-progr. intrinsèques $\Longrightarrow$ risque d'homonimie 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* :
    - ▷ OK en Fortran 90 car au niveau de l'*open*, `PAD="YES"` pris par défaut.
    - ▷ Erreur en Fortran 77 !

## 1.3 – Introduction : apports



- 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.



## 1.4 – Introduction : aspects obsolètes



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. Boucles `DO` pilotées par réels : `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`

*((H.N.95))* : aspects devenant **Hors Norme 95**.



## 1.4 – Introduction : aspects obsolètes



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

```
                          CHARACTER(7),DIMENSION(4)::CFMT
ASSIGN 2 TO NFMT          I=2
2 FORMAT (F10.2)    ⟹    CFMT(2)='(F10.2)'
PRINT NFMT,TRUC           PRINT CFMT(I),TRUC
```

8. `RETURN` multiples

```
      CALL SP1(X,Y,*10,*20)
      ..
   10 .........
      ..
   20 .........
      ..
      SUBROUTINE SP1(X1,Y1,*,*)
      ..
      ..
      RETURN 1
      ..
      RETURN 2
      ..
```

   $\Longrightarrow$ `SELECT CASE` sur la valeur d'un argument retourné

9. `PAUSE 'Montez la bande 102423 SVP'` *((H.N.95))*
   $\Longrightarrow$ `READ` qui attend les données
10. `FORMAT(9H A éviter)` *((H.N.95))*
    $\Longrightarrow$ Constante littérale : `FORMAT(' Recommandé')`

*((H.N.95))* : aspects devenant **Hors Norme 95**.



## 1.4 – Introduction : aspects obsolètes



**Aspects obsolètes introduits par 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))`.

## 1.5 – Introduction : évolution



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

**Principales nouveautés de la norme 95**

- `FORALL (i=1:n, j=1:m, y(i,j)/=0.) x(i,j)=1./y(i,j)` (cf. Annexe H3 page 254).
- 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 H page 249).
- Fonction intrinsèque `NULL()` pour forcer un pointeur à l'état nul y compris lors de sa déclaration (cf. chap. 7.1 page 110).
- Libération automatique des tableaux dynamiques locaux n'ayant pas l'attribut `SAVE` (cf. chap. 6.2 page 107).
- Valeur initiale par défaut pour les composantes d'un type dérivé (cf. chap. 3.1 page 46).
- Fonction intrinsèque `CPU_TIME` (cf. chap. 14.6 page 191).
- Bloc `WHERE` : imbrication possible (cf. chap. 5.6 page 102).
- Expressions d'initialisation étendues (cf. Annexe E page 243).
- `MAXLOC`/`MINLOC` : ajout argument `dim` (cf. Chap. 5.5.1 page 82).
- Suppressions (cf. page 14)... et nouveaux aspects obsolètes.



## 1.6 – Introduction Fortran 90 sur IBM-RS/6000



- Appel du système de compilation
  - **f77** : compatibilité maximale avec anciennes versions ⟹ (`-qfixed`)
  - **f90** : conformance Fortran 95 ⟹ (`-qfree`)
- Quelques options :
  - `-qlanglvl=77std/90std/95std/95pure/extended`
  - `-qsource/nosource`, `-qlistopt`
  - `-C` ⟹ *bounds checking*
  - `-O1/2/3/4` ⟹ Optimisation de niveau 1/2/3/4
  - `-Q` ⟹ insertion automatique (*inlining*)
  - `-qfixed` ou `-qfree`
  - `-qdbg` ou `-g` ⟹ débogueur symbolique **dbx**
  - `-Wl,` *opt-loader* ⟹ **ld**
- Directives par `@PROCESS`
  Action prioritaire sur les options de la ligne de commande.

Pour plus d'informations :

- `http://www.idris.fr/su/Scalaire/Page_generale.html`

Vous y trouverez des FAQ, des supports de cours, la documentation IBM et toute information utile sur l'exploitation des machines scalaires (Pascal et Tlon) de l'IDRIS.



## 1.7 – Introduction Fortran 90 sur NEC SX-5



- Appel du système de compilation :
  - `f90 prog.f` : pour compiler sur SX-5 en "format fixe",
  - `f90 prog.f90` : pour compiler sur SX-5 en "format libre",
  - `sxf90 prog.f90` : pour compiler sur la frontale avec le *cross-compiler* en "format libre".
- Quelques options :
  - `-R5` ⟹ listage indenté avec infos sur vectorisation
  - `-eC` ⟹ débordements de tableaux
  - `-Pi line=100 nest=2` ⟹ insertion automatique (*inlining*)
  - `-Wf"-L summary"` ⟹ liste des options activées
- Directives par : `!CDIR`
  Action prioritaire sur les options de la ligne de commande.
- Bibliothèques mathématiques :
  - LAPACK, BLAS, EISPACK, ScaLAPACK, ASL, Mpack, ...
  - NAG, IMSL.
- Outils d'analyse, de débogage et de conversion :
  - `Psuite`

Pour plus d'informations :

- `http://www.idris.fr/nec/index-nec.html`



## 1.8 – Introduction : bibliographie



**Bibliographie :**


- Aberti C., *Fortran 90, Initiation à partir de Fortran 77*, Série Informatique, 1992, (144 pages), ISBN 2-909615-00-6.
- Adams, Brainerd, Martin, Smith et Wagener, - *Fortran 90 Handbook*, Mc. Graw-Hill, 1992, (740 pages), ISBN 0-07-000406-4. - *Fortran 95 Handbook*, MIT Press, 1997, (711 pages), ISBN 0-262-51096-0.
- Brainerd, Goldberg, Adams, *Programmer's guide to Fortran 90*, 3e 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.

## 1.8 – Introduction : bibliographie




- LIGNELET P., *Structures de données et leurs algorithmes avec Fortran 90 et Fortran 95*, MASSON, 1996, (360pages), ISBN 2-225-85373-8.
- 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.
- 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.




## 1.9 – Introduction : documentation



- Documentation IBM RS/6000 :
  - *XL Fortran 6.1 Language Reference*
  - *XL Fortran 6.1 USER's Guide*
  - *ESSL - Engineering and Scientific Subroutine Library Guide*

  Disponibles sur le serveur Web IDRIS à l'URL :
  `http://www.idris.fr/data/doc_fournisseur/ibm/index-ibmdoc.html`

- Documentation IDRIS RS/6000 :
  - descriptif matériel et logiciel,
  - supports de cours,
  - FAQ,
  - etc.

  Disponibles sur le serveur Web IDRIS à l'URL :
  `http://www.idris.fr/su/Scalaire/Page_generale.html`



## 1.9 – Introduction : documentation



- Documentation NEC SX-5
  - *Fortran 90/SX Language Reference Manual* – G1AF06E-7
  - *Fortran 90/SX User's Guide* – G1AF07E-8
  - *Fortran 90/SX Multitasking User's Guide* – G1AF08E-7
  - *SUPER-UX PSUITE User's Guide* – G1AF26E-5
  - *SUPER-UX OpenGL Programmer's Guide* – G1AF24E-1A
  - *SUPER-UX DBX User's Guide* – G1AF19E-5
  - Serveur Web IDRIS :
    - documentation NEC en ligne (accès restreint) :
      `http://www.idris.fr/users/doc_nec-users/glblcont.html`
    - support de cours "Portage de codes sur NEC SX-5" :
      `http://www.idris.fr/su/divers/SX5_p.html`
    - FAQ : `http://www.idris.fr/faqs/neq.html`

- Documentation générale
  - Support de cours Fortran 95 IDRIS
    ⟹ `http://www.idris.fr/data/cours/lang/f90/`
  - *Fortran Market Index*
    ⟹ `http://www.fortran.com/fortran/`



## Cours Fortran 95

# 2 Généralités

## 2.1 Structure d'un programme

## 2.2 Éléments syntaxiques

### 2.2.1 Les identificateurs

### 2.2.2 Le "format libre"

### 2.2.3 Les commentaires

### 2.2.4 Le "format fixe"

### 2.2.5 Les déclarations

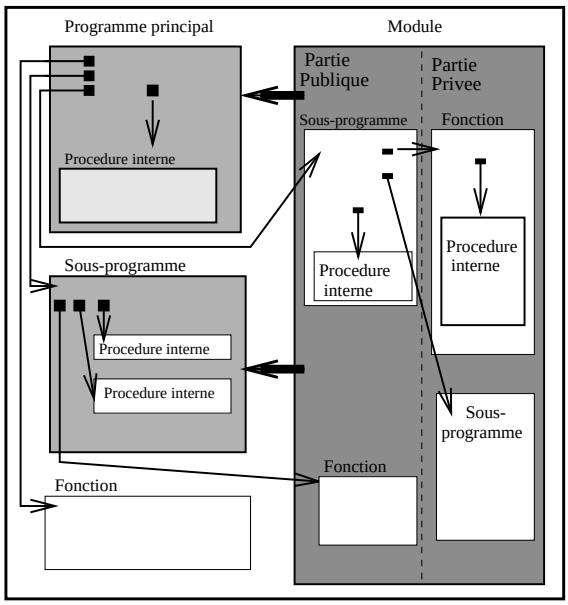
### 2.2.6 Typage et précision des nombres : paramètre KIND

## 2.3 Compilation, édition des liens, exécution





PROGRAMME








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.





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 – Généralités : syntaxe
## 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.



## 2.2.3 – 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 2e instruction ferait partie du commentaire.

**Exemple :**

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

**Attention** :

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

ne sont pas des commentaires Fortran 90 en "**format libre**" et génèrent des erreurs de compilation.



## 2.2.4 – Généralités : syntaxe
## 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 – Généralités : syntaxe
## Les déclarations



**Forme générale d'une déclaration**

**type[, liste_attributs ::] liste_objets**

**Différents types :**

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

## 2.2.5 – Généralités : syntaxe
## Les déclarations



**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 |



## 2.2.5 – Généralités : syntaxe
## Les déclarations



**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
real, dimension(:), pointer :: ptr
real, pointer :: ptr(:) ! non recommandé
```

On *peut* donner le type et les différents attributs d'un objet sur plusieurs instructions (non recommandé) :

**Exemple :**

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



## 2.2.5 – Généralités : syntaxe
## Les déclarations



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 – Généralités : syntaxe
## La précision des nombres (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/6000 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.
(⟹ Voir tableau des sous-types sur IBM RS/6000 et NEC SX5 en annexe A page 196)





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. |





**Fonction intrinsèque `SELECTED_INT_KIND(r)`**

Cette fonction 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)`**

Cette fonction admet deux arguments **p** et **r** indiquant respectivement la **précision** (nombre de chiffres décimaux significatifs) et l'**étendue** (*range*) désirées. Elle retourne une valeur entière (`kind`) qui correspond au **sous-type** permettant de représenter les **réels x** répondant à la demande avec :

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

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

## 2.2.6 – Généralités : syntaxe
## La précision des nombres (kind)



**Exemple :**

```
integer, parameter   :: prec = &
                        selected_real_kind(p=12, 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) ! Certainement 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.



## 2.2.6 – 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._8)` | 2465 | 307 |
| `precision(1._8)` | 13 | 15 |
| `range(1.0)` | 2465 | 37 |
| `precision(1.0)` | 13 | 6 |
| `range(1.d0)` | 2465 | 307 |
| `precision(1.d0)` | 28 | 15 |



## 2.3 – Généralités : compilation, édition de liens, exécution



**Sur NEC SX-5 et IBM RS/6000**

- 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/6000, 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 -qsource 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`

**Note** : il est recommandé d'utiliser le cross-compilateur NEC `sxf90` sur la frontale (Rhodes à l'IDRIS) du SX5 pour les étapes 1 et 2.



## Cours Fortran 95

# Cours Fortran 95
# 3 – Types dérivés : plan



## 3 Types dérivés


### 3.1 Définition et déclaration de structures

### 3.2 Initialisation (constructeur de structure)

### 3.3 Symbole % d'accès à un champ

### 3.4 Types dérivés et procédures

### 3.5 Conclusion et rappels






# 3.1 – Types dérivés : définition

- **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 – Types dérivés : initialisation



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 – Types dérivés : symbole % d'accès à un champ



- `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.
- Voici une nouvelle définition de la couleur jaune (Rouge + Vert) :

```
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)
```

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 tous les deux l'attribut `SEQUENCE` (pour assurer un stockage des champs dans le même ordre et suivant les 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
..........
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 ! >>>> avec type dérivé imbriqué
    integer     :: n  ! nombre de vecteurs
    type(VECTEUR),dimension(20) :: vect !taille max.
  end type CHAMP_VECTEURS
!-------------------------Déclarations ---------
  type(VECTEUR)               :: u,v,w
  type(CHAMP_VECTEURS)        :: champ
  real                        :: ps
!-----------------------------------------------
  u=vecteur(1.,0.,0.)  ! >>>>> Constructeur 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
```





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

```
program testmat
  implicit none
  type MATRICE
    integer :: N, M  ! >>>> Matrice ( N x M )
    real, dimension(:,:),allocatable :: A !>>>> ERREUR
  end type MATRICE
  !----------------------Déclarations ---------
  type(MATRICE)               :: 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(MATRICE),intent(in) :: mat1,mat2
    type(MATRICE)            :: som_mat
    if (mat1%M /= mat2%M .or. mat1%N /= mat2%N) then
      stop ' ERREUR: matrices de 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).





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 90 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 2000**.

## 3.5 – 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 re-définition du type est nécessaire.
- un champ peut avoir l'attribut `POINTER` mais pas `TARGET`.





## 3.5 – 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 121 ;
- la gestion de *listes chaînées* basées sur des types derivés tels :

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

  cf. exemple simple en annexe F 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 sauf pour celles ayant l'attribut `pointer` pour lesquelles il réalise une association.





## 3.5 – Types dérivés : conclusion

*Pages réservées pour vos notes personnelles...*





## Cours Fortran 95

# Cours Fortran 95
# 4 – Programmation structurée : plan



## 4 Programmation structurée


### 4.1 Introduction

### 4.2 Boucles DO

### 4.3 Construction SELECT-CASE




# 4.1 – Programmation structurée
# Introduction



Structure habituelle d'un programme en blocs :

**Exemple :**

```
IF (expression logique) THEN
     bloc1
ELSE IF (expression logique) THEN
       bloc2
     ELSE
       bloc3
END IF
```



# 4.2 – Programmation structurée
# 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
```



# 4.2 – 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
```

**3e 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
```





**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
```





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
```





**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
```

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

**Exemple :**

```
integer :: mois, nb_jours
logical :: annee_bissect
     ........
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_bissect)
         case(.true.)
           nb_jours = 29
         case(.false.)
           nb_jours = 28
    end select fevrier
  case default
    print *, ' Numéro de mois invalide'
end select
```










## 5 Extensions tableaux

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

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

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

**5.4 Section de tableau non contiguë en argument d'une procédure**

**5.5 Fonctions intrinsèques tableaux**

**5.5.1 Interrogation** (`maxloc, lbound, shape,..` )

**5.5.2 Réduction** (`all, any, count, product,...`)

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

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

**5.6 Instruction et bloc `WHERE`**






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**.

## 5.1 – Extensions tableaux Définitions



**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 – Extensions tableaux Manipulations de tableaux



**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
```



## 5.2 – Extensions tableaux Manipulations de tableaux



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.



## 5.2 – Extensions tableaux Manipulations de tableaux



**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 /)
```

**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.

**Exemples :**

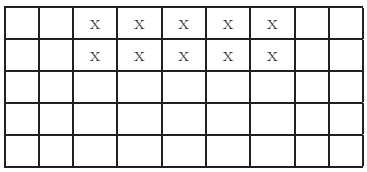
```
integer, dimension(10) :: a = (/ (i, i=1,10) /)
integer, dimension(6)  :: b
integer, dimension(3)  :: c
c(:)     = a(3:10:3)  ! <== "Gather"
b(1:6:2) = c(:)       ! <== "Scatter"
```



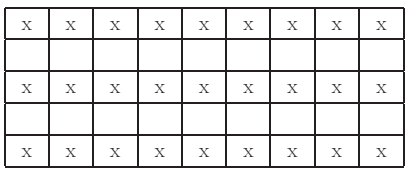


```
integer, dimension(5,9) :: a
```

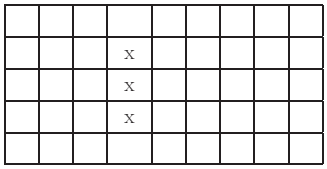
a(1:2,3:7)



a(1:5:2,:)



a(2:4,4)







```
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
```





**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



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

**Exemple :**

```
integer, parameter      :: n=5,m=6
integer, dimension(n,m) :: t
t = 0
call sp(t,n,m)
end

subroutine sp(t,n,m)
integer                 :: n,m
integer, dimension(n,m) :: t
print *,t
end
```

En fait, 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. C'est pour cela qu'il est possible de préciser le caractère * à la place de la dernière dimension.
À l'exécution de la procédure les **étendues** de chaque dimension, hormis la dernière, seront connues. Ce qui implique que la **taille** du tableau ne sera pas connue, et que c'est donc au développeur de s'assurer qu'il n'y a pas de débordement. Ce type de tableau s'appelle tableau à **taille implicite** (*assumed-size-array*).



## 5.3 – 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 2e argument optionnel qui permet de préciser la dimension suivant laquelle on désire connaître le nombre d'éléments. Dans ce cas, elle retourne en fait l'**étendue** relativement à la dimension spécifiée.
Dans l'exemple précédent, il est possible de récupérer l'**étendue** de toutes les dimensions du tableau **t** à l'exception de la dernière.



## 5.3 – 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 effectuer certaines déductions (sert notamment à 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*).



## 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...

**⟹ 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).
- **sur RS6000**, l'argument muet n'a pas l'attribut `target`. En effet, le compilateur IBM force la copie du tableau passé si l'argument muet correspondant a l'attribut `target`...

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.

## 5.4 – 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

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



## 5.5.1 – Fonctions intrinsèques tableaux : interrogation



`MAXLOC(array[,dim][,mask])` ⟹ `[,dim]` Fortran 95 *only*

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

retournent soit un tableau de rang 1 correspondant à l'**emplacement de l'élément maximum/minimum** du tableau passé en argument, soit un tableau de rang **n-1** si `dim` est spécifié (**n** étant le rang de `array`).

`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.

`SIZE(array[,dim])`

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

`SHAPE(source)`

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



## 5.5.1 – Fonctions intrinsèques tableaux : interrogation



**Exemples :**

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

| | | |
|---|---|---|
| `UBOUND(t)` | ⟹ | `(/ 27,49 /)` |
| `UBOUND(t,dim=2)` | ⟹ | `49` |
| `LBOUND(t)` | ⟹ | `(/ -2,0 /)` |
| `LBOUND(t,dim=1)` | ⟹ | `-2` |
| `SHAPE(t)` | ⟹ | `(/ 30,50 /)` |
| `SIZE(t)` | ⟹ | `1500` |
| `SIZE(t,dim=1)` | ⟹ | `30` |
| `MAXLOC((/ 2,-1,10,3,-1 /))` | ⟹ | `(/ 3 /)` |
| `MINLOC((/ 2,-1,10,3,-1 /))` | ⟹ | `(/ 2 /)` |



## 5.5.1 – 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(A, mask=A.LT.5)` | ⟹ | `(/ 2,2 /)` |
| `MINLOC(A, mask=A.GT.5)` | ⟹ | `(/ 3,3 /)` |
| `MINLOC(A, mask=A>8 )` | ⟹ | résultat imprévisible |

`DIM=i` ⟹ la fonction travaille globalement sur cet indice pour chaque valeur fixée dans les autres dimensions.

| | | |
|---|---|---|
| `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 /)` |

## 5.5.2 – Fonctions intrinsèques tableaux : réduction



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 pour chaque valeur fixée dans les autres dimensions.

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

- 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)))...`



## 5.5.2 – Fonctions intrinsèques tableaux : réduction



`ANY(mask[,dim])`

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

- 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)))...`



## 5.5.2 – Fonctions intrinsèques tableaux : réduction



`COUNT(mask[,dim])`

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

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

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 /)`



## 5.5.2 – Fonctions intrinsèques tableaux : réduction



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

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

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

`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 /)`

## 5.5.2 – Fonctions intrinsèques tableaux : réduction



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

`PRODUCT((/ 2,5,-6 /))` $\Longrightarrow$ `-60`
`SUM((/ 2,5,-6 /))` $\Longrightarrow$ `1`

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

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



## 5.5.3 – Fonctions intrinsèques tableaux : multiplication



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 $\mathbf{A}\begin{pmatrix} 3 & -6 & -1 \\ 2 & 3 & 1 \\ -1 & -2 & 4 \end{pmatrix}$ et le vecteur $\mathbf{V}\begin{pmatrix} 2 \\ -4 \\ 1 \end{pmatrix}$

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



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

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

Si `profil(a) = (/ n,p /)` $\Longrightarrow$ `profil(c) = (/ n /))`
et `profil(b) = (/ p /)` $c_i$ = `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 – Fonctions intrinsèques tableaux : construction et transformation



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

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

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

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

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

**Remarque** : ce type de fonction est utilisable au niveau des initialisations (cf. annexe E page 243).





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

Cette fonction permet d'effectuer des décalages **circulaires** sur les éléments dans une dimension (`DIM`) donnée d'un tableau.

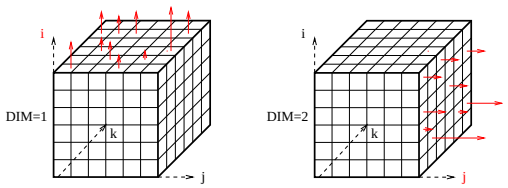
**Par défaut** : `DIM=1`

**Exemples :**

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

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





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

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

Cf. annexe D1 (page 241) pour autre exemple.





`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 |

## 5.5.4 – 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 :

$$\texttt{t1_car} \begin{pmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{pmatrix} \qquad \texttt{t2_car} \begin{pmatrix} d & h & l & p \\ q & r & s & t \\ q & r & s & t \\ q & r & s & t \end{pmatrix}$$

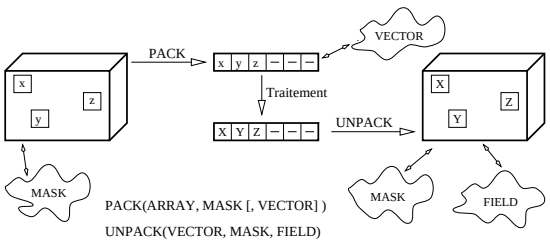


## 5.5.4 – Fonctions intrinsèques tableaux : construction et transformation



`PACK(array,mask[,vector])`

Cette fonction permet de compresser un tableau sous le contrôle d'un masque. Le résultat est un vecteur.




**Exemples :**

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

On obtient : `(/ 9,6 /)`
`(/ 9,6,27,64,125 /))`

- Pour linéariser une matrice : `PACK(a,mask=.true.)`



## 5.5.4 – Fonctions intrinsèques tableaux : construction et transformation



`UNPACK(vector,mask,field)`

Cette fonction décompresse un vecteur sous le contrôle d'un masque.

**Exemples :**

```
logical, dimension(3)   :: v1 = .true.
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 :

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

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

Cf. annexe D2 (page 242) pour autre exemple.



## 5.5.4 – Fonctions intrinsèques tableaux : construction et transformation



`SPREAD(source,dim,ncopies)`

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

**Exemples :**

```
integer, dimension(3) :: a = (/ 4,8,2 /)
print *, SPREAD('a', dim=1, ncopies=3)
print *, SPREAD(a,   dim=2, ncopies=3)
```

On obtient les résultats suivants :

$$\begin{pmatrix} 'a' \\ 'a' \\ 'a' \end{pmatrix} \text{ et } \begin{pmatrix} 4 & 4 & 4 \\ 8 & 8 & 8 \\ 2 & 2 & 2 \end{pmatrix}$$

`MERGE(tsource,fsource,mask)`

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

**Exemple :**

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

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

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

## 5.5.4 – Fonctions intrinsèques tableaux : construction et transformation



`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 – Extensions tableaux 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** :

```
WHERE (mask)
     bloc1
ELSEWHERE
     bloc2
END WHERE
```

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. Le membre de droite de ces affectations doit être un tableau conformant avec le masque.

**Exemple :**

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



## 5.6 – Extensions tableaux Instruction et bloc WHERE



**Ce qui est équivalent à** :

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

**Remarques** :

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

  `WHERE` (expression_logique_tableau) instruction

  **Exemple :** $\Longrightarrow$ `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`.



## 5.6 – 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).

# Cours Fortran 95
# 6 – Gestion mémoire : plan



## 6 Gestion mémoire


### 6.1 Tableaux automatiques

### 6.2 Tableaux dynamiques (ALLOCATABLE, profil différé)




# 6.1 – Gestion mémoire
# 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.

**Exemples :**

```
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 – Gestion mémoire
# Tableaux dynamiques



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

**Exemple :**

```
real, dimension(:,:), allocatable :: a
integer                           :: n,m,err
... . .
read *,n,m
if (.not. allocated(a)) then
  allocate(a(n,m),stat=err)
  if (err /= 0) then
    print *,"Erreur à l'allocation du tableau a"
    stop 4
  end if
end if
... . .
deallocate(a)
... . .
```



# 6.2 – Gestion mémoire
# Tableaux dynamiques



**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 123).
- Un tableau dynamique (*allocatable*) peut avoir l'attribut `TARGET` (cf. chap. 7 suivant). Sa libération (*deallocate*) devra obligatoirement se faire en spécifiant ce tableau et en aucun cas un pointeur intermédiaire lui étant associé.

# Cours Fortran 95
# 7 – Pointeurs : plan



## 7 Pointeurs


7.1 Définition, états d'un pointeur

7.2 Déclaration d'un pointeur

7.3 Symbole =>

7.4 Symbole = appliqué aux pointeurs

7.5 Allocation dynamique de mémoire

7.6 Instruction `NULLIFY`

7.7 Fonction intrinsèque `ASSOCIATED`

7.8 Situations à éviter

7.9 Déclaration de "tableaux de pointeurs"

7.10 Passage en argument de procédure




# 7.1 – Pointeurs
# 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 appelé cible.

**Norme 95** : fonction intrinsèque `NULL()` pour forcer un pointeur à l'état nul (y compris lors de sa déclaration) :

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



# 7.2 – Pointeurs
# 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" !!



# 7.3 – Pointeurs : 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** | ⊗ | ⊗ |

## 7.3 – 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.



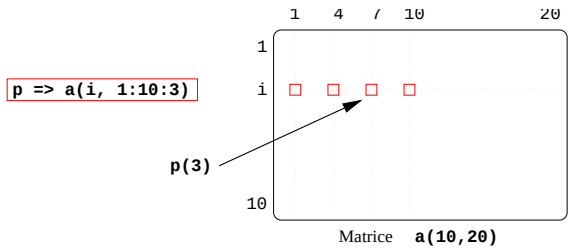
## 7.3 – Pointeurs : symbole =>



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 – Pointeurs
## Symbole = appliqué aux pointeurs



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

**Exemple :**

```
implicit none
integer                         :: i
integer, pointer                :: ptr1, ptr2
integer, target                 :: i1, i2
real, dimension(3,3), target    :: a, b
real, dimension(:,:), pointer :: p, q
!----------------------------------------
i1 = 1 ; i2 = 2
ptr1 => i1
ptr2 => i2
ptr2 = ptr1
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 – Pointeurs
## 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 54 et annexe F1 page 245).

# 7.6 – Pointeurs
# Instruction NULLIFY



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

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



# 7.7 – Pointeurs
# Fonction intrinsèque ASSOCIATED



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

**Syntaxe** :

```
ASSOCIATED(pointeur      [,pointeur])
                         [,cible])
```

| | | |
|---|---|---|
| `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 `c` |
| | –> | faux sinon |

**Remarques :**

- 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 – Pointeurs
# 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.



# 7.8 – 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 – Pointeurs "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
 end type ptr_chaine
end module chaine

program tri_chaine
 use chaine
 implicit none
 type(ptr_chaine), &
     dimension(:), allocatable  :: tab_pointeurs
 type(ptr_chaine)               :: temp
 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))
```



## 7.9 – 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
       temp%p               =>   &
            tab_pointeurs(i)%p
       tab_pointeurs(i)%p   =>   &
            tab_pointeurs(i+1)%p
       tab_pointeurs(i+1)%p => temp%p
       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))
end program tri_chaine
```



## 7.10 – Pointeurs Passage en argument de procédures



**Pointeur passé en argument d'appel d'une 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 80).
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`),



## 7.10 – Pointeurs Passage en argument de procédures



**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 80)...

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.

**Pointeurs et COMMON**

Les attributs `target` et `pointer` sont autorisés pour les entités déclarées dans un `COMMON` (C.f. annexe F2 page 246).

# Cours Fortran 95
## 8 – Interfaces de procédures et modules : plan



## 8 Interface de procédures et modules


8.1 Interface “implicite” : définition

8.2 Interface “implicite” : exemple

8.3 Arguments : attributs INTENT et OPTIONAL

8.4 Passage d’arguments par mot-clé

8.5 Interface “explicite” : procédure interne (CONTAINS)

8.6 Interface “explicite” : 5 cas possibles

8.7 Interface “explicite” : bloc interface

8.8 Interface “explicite” : ses apports

8.9 Interface “explicite” : module avec bloc interface (USE)

8.10 Interface “explicite” : module avec procédure

8.11 Cas d’interface “explicite” obligatoire




## 8.1 – Interface de procédures et modules
## 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-programme 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 de procédures et modules
## Interface “implicite” : exemple



```
program inout
  implicit none
  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,val_max,val_min,ctl,rgmax)
  real,dimension(n)  :: vect
  ..........
  V=val_max+....! Erreur si val_max en sortie seult.
  n=....        ! Erreur si n en entrée seulement
  ctl=99        ! Erreur si 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 ...
     ```

## 8.4 – Interface de procédures et modules Passage d'arguments par mot-clé



**Arguments à mots-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,val_max=vmax,val_min=vmin,&
                          ctl=ctl,rgmax=rgmax)
call maxmin(v,vmax,vmin,ctl=ctl)
call maxmin(v,vmax,ctl=ctl,val_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,val_min=vmin)
  ..............
contains
subroutine maxmin(vect,val_max,val_min,ctl,rgmax)
  implicit none
  integer,dimension(1)            :: rg  ! Attention
  real,dimension(:),intent(in)    :: vect
  real,             intent(out)   :: val_max,val_min
  integer, optional,intent(out)   :: rgmax
  integer,          intent(inout) :: ctl
  val_max=MAXVAL(vect); val_min=MINVAL(vect);  ctl=1
  if(present(rgmax))then !fct. intrins. type logique
    rg=MAXLOC(vect); rgmax=rg(1); ctl=2
  endif
end subroutine maxmin
end program inout
```



## 8.6 – Interface "explicite" 5 cas possibles



**Remarque :**

si la solution **interface "explicite"** par **procédure interne** est simple et permet bien de résoudre à la compilation tous les cas d'erreurs signalés, elle présente des inconvénients qui en limitent l'utilisation :

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

Il existe d'autres solutions pour profiter de la fiabilité associée à l'interface "explicite". Voici les 5 cas d'interface "explicite" :

1. procédures intrinsèques (Fortran 77 et Fortran 90),
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.

Les cas 3, 4 et 5 seront exploités ci-après sur le 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** :

## 8.7 – 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,val_max,val_min,ctl,rgmax)
     real,dimension(:),intent(in)   :: vect
     real,             intent(out)  :: val_max,val_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,val_max,val_min,ctl,rgmax)
 implicit none
 integer,dimension(1)              :: rg
   real,dimension(:),intent(in)    :: vect
   real,             intent(out)   :: val_max,val_min
   integer, optional,intent(out)   :: rgmax
   integer,          intent(inout) :: ctl
```



## 8.7 – Interface “explicite” Bloc interface



```
 val_max=MAXVAL(vect)
 val_min=MINVAL(vect)
 ctl=1
 if(present(rgmax))then
   rg=MAXLOC(vect)
   rgmax=rg(1)
   ctl=2
 endif
 print *,' Taille du vecteur via size  :',SIZE(vect)
 print *,' Profil du vecteur via shape :',SHAPE(vect)
end subroutine maxmin
```



## 8.8 – Interface de procédures interface “explicite” : ses apports



**Rappel : ce qu’apporte l’interface “explicite”**

- la transmission du **profil** et de la **taille** des tableaux à profil implicite et la possiblité 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 avec bloc interface



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

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

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

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

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).

## 8.9 – Interface “explicite” Module avec bloc interface



```
module bi_maxmin
  interface
   subroutine maxmin(vect,val_max,val_min,ctl,rgmax)
     real,dimension(:),intent(in)   :: vect
     real,             intent(out)  :: val_max,val_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,val_max,val_min,ctl,rgmax)
    implicit none
    integer,dimension(1)            :: rg
    real,dimension(:),intent(in)    :: vect
    real,             intent(out)   :: val_max,val_min
    integer, optional,intent(out)   :: rgmax
    integer,          intent(inout) :: ctl
    val_max=MAXVAL(vect)
    val_min=MINVAL(vect)
    ctl=1
    if(present(rgmax))then
      rg=MAXLOC(vect); rgmax=rg(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)
  .......
end program inout
```



## 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**.



## 8.11 – 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 !

# Cours Fortran 95
# 9 – Interface générique : plan



## 9 Interface générique


### 9.1 Introduction

### 9.2 Exemple

### 9.3 Exemple avec `module procedure`




# 9.1 – Interface générique
# Introduction



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

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

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



# 9.2 – Interface générique : exemple



**Exemple :**

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 créer :

1. les deux sous-programmes `rmaxmin` et `imaxmin`,
2. un bloc interface *familial* de nom `maxmin` et le stocker dans un module `big1_maxmin` qui sera compilé séparément puis stocké sous forme de *module objet*,
3. un exemple d'utilisation en prenant soin (c'est obligatoire dans ce contexte) de donner accès (via `USE`) au module contenant l'interface générique en tête de toute unité de programme appelant le sous-programme `maxmin`.

**Remarque** : on peut aussi stocker les 2 sous-programmes en tant que procédures internes du module contenant l'interface générique qui est alors simplifiée en utilisant la notion de `MODULE PROCEDURE`.



# 9.2 – Interface générique : exemple



Voici les deux sous-programmes :

```
subroutine rmaxmin(vect,val_max,val_min,ctl,rgmax)
  implicit none
  integer,dimension(1)            :: rg
  real,dimension(:),intent(in)    :: vect
  real,             intent(out)   :: val_max,val_min
  integer, optional,intent(out)   :: rgmax
  integer,          intent(inout) :: ctl
  val_max=MAXVAL(vect); val_min=MINVAL(vect);  ctl=1
  if(present(rgmax))then !fct. intrins. type logique
    rg=MAXLOC(vect); rgmax=rg(1); ctl=2
  endif
end subroutine rmaxmin

subroutine imaxmin(vect,val_max,val_min,ctl,rgmax)
  implicit none
  integer,dimension(1)            :: rg
  integer,dimension(:),intent(in) :: vect
  integer,          intent(out)   :: val_max,val_min
  integer, optional,intent(out)   :: rgmax
  integer,          intent(inout) :: ctl
  val_max=MAXVAL(vect); val_min=MINVAL(vect);  ctl=1
  if(present(rgmax))then !fct. intrins. type logique
    rg=MAXLOC(vect); rgmax=rg(1); ctl=2
  endif
end subroutine imaxmin
```

## 9.2 – Interface générique : exemple 145

**Voici le module et le bloc interface générique** :

```
module big1_maxmin
  interface maxmin
   subroutine rmaxmin(vect,val_max,val_min,ctl,rgmax)
     real,dimension(:),intent(in)   :: vect
     real,             intent(out)  :: val_max,val_min
     integer, optional,intent(out)  :: rgmax
     integer,          intent(inout):: ctl
   end subroutine rmaxmin
!
   subroutine imaxmin(vect,val_max,val_min,ctl,rgmax)
     integer,dimension(:),intent(in):: vect
     integer,          intent(out)  :: val_max,val_min
     integer, optional,intent(out)  :: rgmax
     integer,          intent(inout):: ctl
   end subroutine imaxmin
  end interface
end module big1_maxmin
```

**Norme 95** : le nom générique (ici `maxmin`) spécifié au niveau de l'instruction `interface maxmin` peut être répété au niveau de `end interface maxmin`.



## 146 9.2 – Interface générique : exemple

**Voici le programme utilisant la fonction générique** :

```
program inout
  USE big1_maxmin !<<<--Accès au bloc interface
  implicit none   !                   générique
  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)
  .......
  call sp1(n)
  .......
end program inout

subroutine sp1(n)
  USE big1_maxmin  !<<<--Accès au bloc interface
  implicit none    !                   générique
  integer,intent(in)  :: n
  integer             :: rgmax=0,ctl=0
  integer,dimension(n):: v
  integer             :: vmax,vmin
  .......
  call maxmin(v,vmax,vmin,ctl,rgmax)
  .......
end subroutine sp1
```



## 9.3 – Interface générique Exemple avec `module procedure` 147

**Solution avec "`module procedure`"**

Dans la solution précédente, le module interface générique était indépendant des 2 sous-programmes externes. Là, les deux procédures `rmaxmin` et `imaxmin` font partie du module `big2_maxmin`. De ce fait leur interface est *explicitement* définie et le bloc interface est réduit :

```
module big2_maxmin
    interface maxmin
      module procedure rmaxmin,imaxmin !<<<<<<<<<<<<
    end interface
 contains
 subroutine rmaxmin(vect,val_max,val_min,ctl,rgmax)
  implicit none
  integer,dimension(1)            :: rg
  real,dimension(:),intent(in)    :: vect
  real,             intent(out)   :: val_max,val_min
  integer, optional,intent(out)   :: rgmax
  integer,          intent(inout) :: ctl
  val_max=MAXVAL(vect); val_min=MINVAL(vect);  ctl=1
  if(present(rgmax))then !fct. intrins. type logique
    rg=MAXLOC(vect); rgmax=rg(1); ctl=2
  endif
 end subroutine rmaxmin
!----------------------
```



## 148 9.3 – Interface générique Exemple avec `module procedure`

```
 subroutine imaxmin(vect,val_max,val_min,ctl,rgmax)
  implicit none
  integer,dimension(1)            :: rg
  integer,dimension(:),intent(in) :: vect
  integer,          intent(out)   :: val_max,val_min
  integer, optional,intent(out)   :: rgmax
  integer,          intent(inout) :: ctl
  val_max=MAXVAL(vect); val_min=MINVAL(vect);  ctl=1
  if(present(rgmax))then !fct. intrins. type logique
    rg=MAXLOC(vect); rgmax=rg(1); ctl=2
  endif
 end subroutine imaxmin
end module big2_maxmin
```

**Voici le programme utilisant ce module**:

```
program inout
  USE big2_maxmin !<<<--Accès au bloc interface et
                  !     aux procédures
  real,dimension(n)  :: v=(/ 1.,2.,40.,3.,4. /)
  .......
  call maxmin(v,vmax,vmin,ctl,rgmax)
  call sp1(n)
end program inout
subroutine sp1(n)
  USE big2_maxmin !<<<--Accès au bloc interface et
                  !     aux procédures
  integer,dimension(n) :: v
  .......
  call maxmin(v,vmax,vmin,ctl,rgmax)
end subroutine sp1
```

**Note**: vous trouverez en annexe G (page 247) une application particulière de l'interface générique permettant de fiabiliser l'appel d'une procédure Fortran 77.









## 10 Surcharge ou création d'opérateurs

10.1 Introduction
10.2 Interface operator
10.3 Interface assignment





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.

## 10.1 – Surcharge d'opérateurs Introduction



**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 – Surcharge d'opérateurs Interface operator



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

Dans le cas de la **définition** d'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 matrice
  implicit none
  type matrice
    integer                         :: n,m
    real, dimension(:,:), pointer :: ptr_mat
  end type matrice
  interface operator(+)
    module procedure add
  end interface
contains
  function add(a,b)
    type(matrice), intent(in)   :: a,b
    type(matrice)               :: 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 matrice
```



## 10.2 – Surcharge d'opérateurs Interface operator



```
program appel
  use matrice
  implicit none
  integer       :: err,i,j
  type(matrice) :: a,b,c
  print *,'Entrer la valeur de n :'
  read(*,*)n; a%n = n; b%n = n
  print *,'Entrer la valeur de m :'
  read(*,*)m; a%m = m; b%m = m
  allocate(a%ptr_mat(n,m), stat=err)
  if (err /= 0) then
    print *,'Erreur allocation matrice a'; stop 4
  endif
  allocate(b%ptr_mat(n,m), stat=err)
  if (err /= 0) then
    print *,'Erreur allocation matrice b'; stop 4
  endif
  a%ptr_mat=reshape((/ ((real(i+j),i=1,n),j=1,m) /), &
                    shape=(/ n,m /))
  b%ptr_mat=reshape((/ ((real(i*j),i=1,n),j=1,m) /), &
                    shape=(/ n,m /))
  c = a + b ! <<<<<<<<<<<<<<
  do i=1,c%n
    print *, c%ptr_mat(i,:)
  end do
end program appel
```



## 10.3 – Surcharge d'opérateurs Interface assignment



Pour surcharger le symbole d'**affectation (=)** on utilisera un bloc interface du type `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 matrice
  implicit none
  integer(kind=2), private        :: err
  type matrice
    integer                         :: n,m
    real, dimension(:,:), pointer :: ptr_mat
  end type matrice
  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(matrice), intent(in)   :: a
    type(matrice)               :: 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(matrice), intent(in)  :: a
    i = a%n*a%m
  end subroutine taille_mat
end module matrix
```





```
program appel
  use matrix
  implicit none
  type(matrice) :: a, b, c, d
  integer       :: err, taille_a, taille_b
     ...
  read *, n, m
  allocate(a%ptr_mat(n,m), &
           b%ptr_mat(n,m), &
           c%ptr_mat(n,m))
     ...
!---------------
  taille_a = a
  taille_b = b
!---------------
  d = .tr.c
!---------------
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.



## Cours Fortran 95
## 11 – Contrôle de visibilité : plan
159

11 Contrôle de visibilité

11.1 Introduction

11.2 Instruction PRIVATE et PUBLIC

11.3 Attribut PRIVATE et PUBLIC

11.4 Type dérivé "semi-privé"

11.5 Paramètre ONLY de l'instruction USE



## 11.1 – Contrôle de visibilité : introduction
160

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 – Contrôle de visibilité : 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 – Contrôle de visibilité : 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
```



## 11.4 – Contrôle de visibilité : 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**.



## 11.4 – Contrôle de visibilité : type dérivé "semi-privé"



**Exemples :**

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

## 11.4 – Contrôle de visibilité : type dérivé “semi-privé”



**Exemple** complet de création d'un module au sein duquel on définit :
- des variables globales (alternative au `COMMON`),
- un type-dérivé *semi-privé*,
- certaines ressources *privées*,
- des procédures de surcharge/définition d'opérateurs,
- des *méthodes* (`poubelle`, `imp`, `init`).

```
module matrix
  integer          :: nb_lignes, nb_col
  integer, private :: err
  type matrice
    private
    integer                        :: n,m
    real, dimension(:,:), pointer :: ptr_mat
!--------------==> Fortran 95 <==-------------!
!   integer                        :: n=0, m=0 !
!   real, dimension(:,:), pointer :: &        !
!         ptr_mat => NULL()                   !
!-------------------------------------------!
  end type matrice
  private :: valorisation, add, taille_mat, trans
!--------------------------------
  interface operator(+)
     module procedure add
  end interface
!--------------------------------
  interface operator(.tr.)
    module procedure trans
  end interface
```



## 11.4 – Contrôle de visibilité : type dérivé “semi-privé”



```
  interface assignment(=)
    module procedure taille_mat, valorisation
  end interface
!--------------------------------
contains
!--------------------------------
  subroutine init(a)
  !----------------> Inutile en Fortran 95 !
    type(matrice), intent(out) :: a
    a%n = 0 ; a%m = 0
    nullify(a%ptr_mat)
  end subroutine init
!--------------------------------
  subroutine valorisation(a,t)
    type(matrice),      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(t, (/ a%n, a%m /))
  end subroutine valorisation
```



## 11.4 – Contrôle de visibilité : type dérivé “semi-privé”



```
!--------------------------------
  subroutine poubelle(a)
    type(matrice), 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(matrice), intent(in)   :: a,b
    type(matrice)               :: 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(matrice), intent(in)  :: a
    i = a%n*a%m
  end subroutine taille_mat
```



## 11.4 – Contrôle de visibilité : type dérivé “semi-privé”



```
!--------------------------------
  function trans(a)
    type(matrice), intent(in)   :: a
    type(matrice)               :: 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(matrice), 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
```

## 11.4 – Contrôle de visibilité : type dérivé “semi-privé”



**Exemple d'unité utilisatrice de ce module :**

```
program appel
  use matrix
  implicit none
  integer                       :: i,j,taille
  type(matrice)                 :: a,b,c,d
  call init(a);call init(b);call init(c);call init(d)
  !-----------> call init inutile en Fortran 95 !
  print *,"Nb. de lignes : ";  read *, nb_lignes
  print *,"Nb. de colonnes :"; read *, nb_col
  a=(/ ((real(i+j),i=1,nb_lignes),j=1,nb_col) /)
  b=(/ ((real(i*j),i=1,nb_lignes),j=1,nb_col) /)
  a=b     ! Que se passe t'il ?
          ! Que faudrait-il faire ?  ------------------
          ! ====================>>   Cf. annexe C ((*))
  do i=1,1000;...; c = a + b; end do!------------------
  call imp(a) ; call imp(b)
  call poubelle(b)
  taille = c
  call imp(c)
  call poubelle(c)
  d = .tr.a
  call poubelle(a)
  call imp(d)
  call poubelle(d)
end program appel
```

((*)) Annexe C : cf. page 235



## 11.5 – Contrôle de visibilité : paramètre ONLY de l'instruction USE



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

**Exemple :**

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

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



## 11.5 – 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*.





## Cours Fortran 95

# Cours Fortran 95
# 12 – Procédures récursives : plan



## 12 Procédures récursives

### 12.1 Clause RESULT

### 12.2 Exemple



# 12.1 – Procédures récursives
# Clause RESULT



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 – Procédures récursives
# Exemple



**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
  if (n <= 1) then
    fibo = 1
  else
    fibo = fibonacci(n-1) + fibonacci(n-2)
  end if
end function fibonacci
```

**Attention** : la nécessité de devoir faire plusieurs appels récursifs au sein d'une fonction (2 dans l'exemple) peut conduire à des temps d'exécution prohibitifs !





# Cours Fortran 95

# Cours Fortran 95
# 13 – Nouveautés sur les E/S : plan



## 13 Nouveautés sur les E/S


**13.1** `OPEN (status, position, action, delim,...)`

**13.2** `INQUIRE (recl, action, iolength, delim,...)`

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

**13.4** **Instruction** `NAMELIST`

**13.5** **Spécification de format minimum**




# 13.1 - Nouveautés sur les E/S : OPEN



`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*.



# 13.1 - Nouveautés sur les E/S : OPEN



`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).

**Exemple :**

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



# 13.2 – Nouveautés sur les E/S : INQUIRE



- `RECL = n` : permet de récupérer la longueur maximale des enregistrements.
- `POSITION = chaîne` : permet de récupérer la valeur du même paramètre spécifié lors de l'**open**.
- `ACTION = chaîne` : permet de récupérer la valeur du même paramètre spécifié lors de l'**open**.
- `DELIM = chaîne` : permet de récupérer la valeur du même paramètre spécifié lors de l'**open**.
- `IOLENGTH=long` : permet de récupérer la longueur de la liste des entités spécifiées. C'est utile lorsque l'on veut valoriser le paramètre `RECL` de l'ordre `OPEN` pour un fichier 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 d'une structure (de type dérivé) faisant partie de la liste spécifiée.

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



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

- 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,
- le paramètre `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=17,end=18)liste
read(8,fmt=9, advance='no', size=n, iostat=icode) liste
```

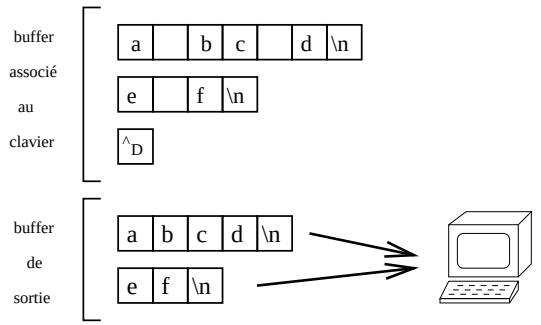


## 13.3 – Nouveautés sur les E/S : entrées-sorties sur les fichiers texte



Exemple :

```
  .......
  character(len=1) :: c
!-------------------------------------------------
! Lecture de lignes au clavier et impression de ces
! 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 – Nouveautés sur les E/S : 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)
.......
```

Les données à lire pourraient se présenter sous l'une des formes suivantes :

```
&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 /
```



## 13.5 – Nouveautés sur les E/S : spécification de format



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

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 : plan



# 14 Quelques nouvelles fonctions intrinsèques


**14.1** Conversions entiers/caractères (`char`, `ichar`,...)

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

**14.3** Manipulations de chaînes (`adjustl`, `index`, `scan`,...)

**14.4** Transformations (`transfer`)

**14.5** Précision/codage numérique (`tiny`, `huge`, `epsilon`, `nearest`, `spacing`,...)

**14.6** Mesure de temps, nombres aléatoires (`date_and_time`, `system_clock`, `random_number`,...)

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




## 14.1 – Fonctions relatives aux chaînes : conversions entiers/caractères



- `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 – Nouvelles fonctions intrinsèques : comparaison de chaînes



- `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 - Nouvelles fonctions intrinsèques : manipulation de chaînes



- `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 - Nouvelles fonctions intrinsèques : transformations



- `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 $\geq 1$, sinon un scalaire.
  - Si `size` présent, retourne un vecteur de taille `size`.

  **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)
```

  qui remplace la version Fortran 77 classique :

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

  **Application** : passer une chaîne à C

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



## 14.5 – Nouvelles fonctions intrinsèques : précision et codage numérique



| | |
|---|---|
| `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. |
| `DIGITS(x)` | en fonction du sous-type de `x`, nombre de chiffres binaires de la mantisse si `x` réel (sur machine IEEE, 24 bits en simple précision, et 52 en double) ou nombre de chiffres binaires de stockage (hors bit de signe) si `x` est entier. |
| `EXPONENT(x)` | entier : valeur de l'exposant du réel `x`. |
| `FRACTION(x)` | réel : valeur de la mantisse du réel `x`. |
| `RANGE(x)` | c.f. chapitre 2 – Généralités (`KIND`). |
| `PRECISION(x)` | c.f. chapitre 2 – Généralités (`KIND`). |



## 14.6 – Nouvelles fonctions intrinsèques : mesure de temps, nombres aléatoires



`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 fourni optionnellement par `zone`. `values` est un vecteur récupérant les informations précédentes sous forme d'entiers.

`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 (cf. `man`).

**Norme 95** : sous-progr. `CPU_TIME(time)` $\Longrightarrow$ temps `CPU` en secondes (ou réel $< 0$ si indisponible).



## 14.7 – Nouvelles fonctions intrinsèques : opérations sur les bits



`IAND(i,j)` fonction retournant un entier de même type que `i` résultant de la combinaison bit à bit de `i` et `j` par un **ET logique**.

`IEOR(i,j)` fonction retournant un entier de même type que `i` résultant de la combinaison bit à bit de `i` et `j` par un **OU exclusif logique**.

`IOR(i,j)` fonction retournant un entier de même type que `i` résultant de la combinaison bit à bit de `i` et `j` par un **OU inclusif logique**.

`ISHFT(i,shift)` fonction retournant un entier de même type que `i` résultant du **décalage** de `shift` bits appliqué à `i`. Décalage vers la gauche ou vers la droite suivant que l'entier `shift` est positif ou négatif. Les bits sortant sont perdus et le remplissage se fait par des zéros.

`ISHFTC(i,shift[,size])` fonction retournant un entier de même type que `i` résultant d'un **décalage circulaire** de `shift` positions appliqué aux `size` bits de droite de `i`. Décalage vers la gauche ou vers la droite suivant que l'entier `shift` est positif ou négatif.

## 14.7 – Nouvelles fonctions intrinsèques : opérations sur les bits



`IBCLR(i,pos)` fonction retournant un entier identique à `i` avec le `pos`$^{\text{ième}}$ bit mis à zéro.

`IBSET(i,pos)` fonction retournant un entier identique à `i` avec le `pos`$^{\text{ième}}$ bit mis à 1.

`NOT(i)` fonction retournant un entier de même type que `i`, ses bits correspondant au **complément logique** de ceux de `i`.

`IBITS(i,pos,len)` fonction stockant dans un entier de même type que `i` les `len` bits de `i` à partir de la position `pos`. Ces bits sont cadrés à droite et complétés à gauche par des zéros.

`MVBITS(from,frompos,len,to,topos)` sous-programme copiant une séquence de bits depuis une variable entière (`from`) vers une autre (`to`).

**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`".







## Cours Fortran 95 Annexe A



# A Annexe : paramètre KIND et précision des nombres

## A.1 Sur IBM RS/6000

## A.2 Sur NEC SX5



## Annexe A1 – Paramètre KIND : précision des nombres sur RS/6000



- **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** : $1.2 \times 10^{-38} \leq |\, r \,| \leq 3.4 \times 10^{38}$
    6 chiffres significatifs décimaux.
  - kind = **8** $\Longrightarrow$ 8 **octets** : $2.2 \times 10^{-308} \leq |\, r \,| \leq 1.8 \times 10^{308}$
    15 chiffres significatifs décimaux.
  - kind = **16** $\Longrightarrow$ 16 **octets** : $2.2 \times 10^{-308} \leq |\, r \,| \leq 1.8 \times 10^{308}$
    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`

## Annexe A2 – Paramètre KIND : précision des nombres 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`







## Cours Fortran 95 Annexe B



# B Annexe : exercices

## B.1 Exercices : énoncés

## B.2 Exercices : corrigés



## Annexe B1 – 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.

Vous pouvez 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, ....

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

Le programme imprimera la matrice obtenue ligne par ligne afin de vérifier son contenu.

**Exercice 3 :**

Compiler et exécuter le programme contenu dans les fichiers `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 (mot ou suite de mots séparés par des blancs) 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)` (en ignorant les blancs éventuels) pour faciliter sa manipulation.
La chaîne pourra être lue au clavier (commencer par l'entrée d'un simple mot) ou lue dans un fichier. Le fichier `palindrome` contient des phrases vous permettant de tester la version finale de votre programme.

**Exercice 5 :**

Compléter le programme contenu dans le fichier `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 165) du support de cours. Il est stocké (avec un programme principal d'utilisation complet) dans le fichier `exo7.f90`.

Complétez le module `matrix` en définissant un opérateur `.vp.` permettant le calcul des valeurs propres d'un objet de type matrice. 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 fichier `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 fichier (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 ou *"news IMSL@RHODES+UQBAR"* sur la frontale Rhodes.

**Exercice 8 :**

Soit le programme principal contenu dans le fichier `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, définir :

- le type `musicien` et un tableau `tab_mus` de ce type (dimensionné à 30).
- le sous-programme `init` devant lire le contenu du fichier `musiciens` (ce fichier contient une liste de compositeurs avec leurs années de naissance et de mort : éditez-le au préalable afin de connaître son formatage) afin de valoriser le tableau `tab_mus` et l'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 :

# Annexe B1 – Exercices : énoncés

```
 ---- 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
```





# Annexe B1 – Exercices : énoncés

```
 ---- 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 fichier `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.





# Annexe B2 – Exercice 1 : 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
  !
  ! Une fois les dimensions connues, on peut allouer la matrice
  !
  allocate(mat(n,m),stat=err)
  if (err /= 0) then
    print *,"Erreur d'allocation"; stop 4
  endif
  !
  ! Remplissage des lignes de rang pair avec l'entier 1
  !
  mat(2:n:2,:) = 1
  !
  ! Remplissage lignes de rang impair 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
```

# Annexe B2 – Exercice 3 : corrigé



**Exercice 3 : corrigé**

**Sur NEC-SX5 (Uqbar) et sa frontale (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 RS/6000 (Pascal ou Tlon)**

```
IBM-RS/6000> f90 -c mod1_exo3.f90 mod2_exo3.f90
IBM-RS/6000> f90 exo3.f90 mod1_exo3.o mod2_exo3.o -o exo3
IBM-RS/6000> exo3
```

**Exemple de makefile sur RS/6000** :

```
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)
        $@
```



# Annexe B2 – Exercice 4 : corrigés



**Exercice 4 : corrigé**

```
program exo4
  !
  !--> Cas de la lecture au clavier d'un simple mot.
  !--> L'entrée de la chaîne "fin" arrête le programme.
  !
  implicit none
  integer, parameter                :: long=10
  integer                           :: ln
  character(len=long)               :: chaine
  character(len=1), dimension(long) :: tab_car
  do
    print *,"Entrez une chaine :" ; read(*,'(a)')chaine
    !
    ! Si chaîne = "fin" on a fini
    !
    if (chaine == "fin") exit
    !
    ! On récupère la longueur utile de la chaîne
    ! entrée (sans les blancs de fin).
    !
    ln = len_trim(chaine)
    !
    ! Copie chaîne entrée dans un tableau de caractères.
    !
    tab_car = transfer(chaine,'a',long)
    !
    ! On compare les éléments avec ceux de la chaîne inversée.
    ! La fonction ALL sert à comparer le contenu de deux tableaux.
    !
    if (all(tab_car(:ln) == tab_car(ln:1:-1))) then
      print *,chaine(:ln)," est un palindrome"
    else
      print *,chaine(:ln)," n'est pas un palindrome"
    endif
  end do
 end program exo4
```



# Annexe B2 – Exercice 4 : corrigés



**Autre solution**, avec lecture du fichier `palindrome` :

```
program exo4_bis
  !---------------
  implicit none
  integer, parameter              :: long=100
  integer                         :: ln, ln_util, eof, i
  character(len=long)             :: chaine
  character(len=1),dimension(long):: tab_car,tab_car_pack
  logical, dimension(long)        :: mask
  !
  open(unit=1, file="palindrome", form="formatted", action="read")
  read(1, '(a)', iostat=eof)chaine
  do while (eof == 0)
                                  ! Longueur utile de la chaîne
    ln_util = len_trim(chaine) ! débarassée de ses blancs de fin.
    !
    ! Copie de la chaîne dans un tableau de caractères.
    !
    tab_car = (/ (chaine(i:i), i=1,long) /)
    !
    ! Compression du tableau "tab_car"
    !
    mask = tab_car /= ' '
    ln = count(mask)
    tab_car_pack(1:ln) = pack(tab_car, mask)
    !
    ! Comparaison des éléments symétriques/au milieu de la chaîne.
    ! La fonction ALL sert à comparer le contenu de deux tableaux.
    !
    if(all(tab_car_pack(:ln/2)==tab_car_pack(ln:ln-ln/2+1:-1))) then
      print *,chaine(:ln_util)," est un palindrome"
    else
      print *,chaine(:ln_util)," n'est pas un palindrome"
    endif
    read(1, '(a)', iostat=eof) chaine
  end do
end program exo4_bis
```



# 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="('Entrez l''ordre du triangle : ')")
    read(5, *)n
    if (n >= 20) exit        ! On limite à 19 lignes (cf. format).
    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 des 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
```





# Annexe B2 – Exercice 7 : corrigé

**Exercice 7 : corrigé**

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

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

  private  :: add, trans, taille_mat, valorisation, affect
  private  :: ramasse_miettes
  !---------------------------------
  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
  !---------------------------------
  subroutine init(a)
    !----------------> Inutile en Fortran 95 !
    type(matrice), intent(out) :: a
    a%n = 0 ; a%m = 0 ; nullify(a%ptr_mat)
  end subroutine init
  !---------------------------------
```



# Annexe B2 – Exercice 7 : corrigé



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

    temp%ptr_mat => a%ptr_mat
    call poubelle(temp)
  end subroutine ramasse_miettes
  !---------------------------------
  subroutine poubelle(a)
    type(matrice), intent(inout) :: a
     ....
  end subroutine poubelle
  !---------------------------------
  function add(a,b)
    type(matrice), intent(in)   :: a,b
    type(matrice)               :: add
     ....
  end function add
  !---------------------------------
  function trans(a)
    type(matrice), intent(in)   :: a
    type(matrice)               :: trans
     ....
  end function trans
  !---------------------------------
  function val_propres(a)
    type(matrice), intent(in)     :: a
    complex, dimension(nb_lignes) :: val_propres
    if (associated(a%ptr_mat)) then
      call evlrg(nb_lignes, a%ptr_mat, nb_lignes, val_propres)
    else
      Stop "Objet non existant"
    end if
  end function val_propres
```





# Annexe B2 – Exercice 7 : corrigé

```
  subroutine taille_mat(i,a)
    integer,       intent(out) :: i
    type(matrice), intent(in)  :: a
    i = a%n*a%m
  end subroutine taille_mat
  !---------------------------------
  subroutine affect(a,b)
    type(matrice), intent(inout) :: a
    type(matrice), intent(in)    :: b
     ....
  end subroutine affect
  !---------------------------------
  subroutine imp(a)
    type(matrice), 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(matrice)   :: a
  real,    dimension(:), allocatable :: val_init
  complex, dimension(:), allocatable :: val_pr
  open(unit=10, file="exo7.data", form='formatted', action='read')
  call init(a) ! <=== inutile en Fortran 95
  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)
  a = val_init
  !---------------
   val_pr = .vp. a
  !---------------
  if (erreur()) then
    print *,"La méthode diverge"
  else
    call imp(a)
    call imp_vp(val_pr)
  end if
  call poubelle(a)
end program exo7
```





# Annexe B2 – Exercice 8 : corrigé

**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 :: tab_ptr_musicien
  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.
    ! N.B.: 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)
    !
    ! Cette procédure trie la liste des musiciens soit par ordre
    ! alphabétique des noms soit 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
    type(ptr_musicien) :: temp
    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
          temp                 = tab_ptr_musicien(i)
          tab_ptr_musicien(i)  = tab_ptr_musicien(i+1)
          tab_ptr_musicien(i+1) = temp
          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 permettant de surcharger l'opérateur < afin
    !-- de pouvoir spécifier des opérandes de type musicien.
    !
    type(musicien), intent(in) :: mus1,mus2
    logical                    :: longevite
    integer                    :: duree_de_vie_mus1, &
                                  duree_de_vie_mus2
    duree_de_vie_mus1 = mus1%annee_mort - mus1%annee_naiss
    duree_de_vie_mus2 = mus2%annee_mort - mus2%annee_naiss
    longevite = duree_de_vie_mus1 < duree_de_vie_mus2
  end function longevite
  !---------------------
  subroutine mort_le_plus_jeune(mus,tab_mus)
    !-- Procédure de 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
      !
      ! N.B.: 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
```



## Annexe B2 – Exercice 9 : corrigés



**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
```



## Annexe B2 – Exercice 9 : corrigés



```
  subroutine tri(critere)
    !
    ! Cette procédure trie la liste des musiciens soit par ordre
    ! alphabétique des noms soit 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 permettant de surcharger l'opérateur <
    ! afin de pouvoir spécifier des opérandes de type musicien.
    !
    type(musicien), intent(in) :: mus1,mus2
    logical                    :: longevite
    integer                    :: duree_de_vie_mus1, &
                                  duree_de_vie_mus2
    duree_de_vie_mus1 = mus1%annee_mort - mus1%annee_naiss
    duree_de_vie_mus2 = mus2%annee_mort - mus2%annee_naiss
    longevite = duree_de_vie_mus1 < duree_de_vie_mus2
  end function longevite
```





```
  function mort_le_plus_jeune(debut)
    type(musicien), intent(in) :: debut
    type(musicien)             :: mort_le_plus_jeune
    type(musicien), pointer    :: p_mus

    mort_le_plus_jeune = debut; p_mus => debut%ptr
    do while(associated(p_mus))
      !
      ! N.B.: 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
```

```
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)
  !
  ! Cette procédure trie la liste des musiciens soit par ordre
  ! alphabétique des noms soit 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
```





```
      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 permettant de surcharger l'opérateur <
  ! afin de pouvoir spécifier des opérandes de type musicien.
  !
  type(musicien), intent(in) :: mus1,mus2
  logical                    :: longevite
  integer                    :: duree_de_vie_mus1, &
                                duree_de_vie_mus2
  duree_de_vie_mus1 = mus1%annee_mort - mus1%annee_naiss
  duree_de_vie_mus2 = mus2%annee_mort - mus2%annee_naiss
  longevite = duree_de_vie_mus1 < duree_de_vie_mus2
end function longevite
!---------------------------------
function mort_le_plus_jeune(debut)
  type(musicien), intent(in) :: debut
  type(musicien)             :: mort_le_plus_jeune
  type(musicien), pointer    :: p_mus
  mort_le_plus_jeune = debut; p_mus => debut%ptr_suivant
  do while(associated(p_mus))
    !
    ! N.B.: 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
```

## Annexe B2 – Exercice 9 : corrigés

```
  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
```





## Cours Fortran 95



## Cours Fortran 95 : Annexe C
## Zones anonymes dormantes



**À propos de l'exemple du paragraphe 11.4 page 169**

Lors de l'affectation `a = b` (cf. exemple référençé ci-dessus) nous avions posé les questions suivantes :

- **Que se passe-t'il ?**
  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 : `a%ptr_mat => b%ptr_mat` ; `a%ptr_mat` prend donc l'état de `b%ptr_mat` c.-à-d. associé à la même cible. De ce fait la zone mémoire *anonyme* qui était auparavant associée à `a%ptr_mat` ne peut plus être référencée et devient donc une zone dormante encombrante et inutile ! De plus, `a` et `b` 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.



## Cours Fortran 95 : Annexe C
## Zones anonymes dormantes



```
module matrix       !<=== Solution avec redéfinition de
  .......           !     l'affectation (cf. original page (**))
  .......
!---------------------------------
  interface assignment(=)
    module procedure taille_mat, valorisation, affect
  end interface
!---------------------------------
contains
 ........
!---------------------------------
  subroutine affect(a,b)
    type(matrice), intent(inout) :: a
    type(matrice), 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. entre matrices non conformantes"
    else
      allocate(a%ptr_mat(b%n,b%m), stat=err)
      if (err /= 0) &
      stop "Erreur ==> allocation membre de gauche mal passée"
!     Note : dans certains cas il peut être préférable de laisser
!     ----   le compilateur gérer l'erreur pour pouvoir récupérer
!            la "traceback" éventuellement plus informative.
!            Dans ce cas, stat=err ne doit pas être spécifié.
    end if
    a%n = b%n ; a%m = b%m
    a%ptr_mat = b%ptr_mat
  end subroutine affect
!---------------------------------
 ........
end module matrix
```

(**) ⟹ page 169

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

1. `c = a + b + d` ou bien, `c = .tr.(a + b)`
2. `do i=1,n; ....; c = a + b; ....; 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 matrice 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*.

⊳⊳⊳ Voici un extrait de la solution adoptée en version Fortran 95 (cf. page suivante).





```
module matrix            ! <=== Solution avec "ramasse-miettes"
  ........               !      Cf. original page (**)
  ........
  type matrice
    private
    logical                        :: CreeParFonction = .false.
    integer                        :: n,m
    real, dimension(:,:), pointer :: ptr_mat => NULL()
  end type matrice
  ........
  ........
  private :: ramasse_miettes
contains
  subroutine ramasse_miettes(a)
    type(matrice), intent(in) :: a
    type(matrice)             :: temp
    !
    temp%ptr_mat => a%ptr_mat
    call poubelle(temp)
  end subroutine ramasse_miettes
  ........
  ........
```

(**) $\Longrightarrow$ page 169





```
  ........
  ........

function add(a,b)
  ........
  add%CreeParFonction = .true.
  if (a%CreeParFonction) call ramasse_miettes(a)
  if (b%CreeParFonction) call ramasse_miettes(b)
end function add

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. version modifiée page (***)
  ........
  if (b%CreeParFonction) call ramasse_miettes(b)
end subroutine affect

........
........
```

(***) $\Longrightarrow$ page 236

# Cours Fortran 95 : annexe D1
# Fonction CSHIFT : 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)` :

- **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, 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.



# Cours Fortran 90 : annexe D2
# Fonctions PACK/UNPACK : 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
 .....
```



# Cours Fortran 95
# Annexe E : expressions d'initialisation



**Sont autorisés** :

- 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` :
  `INT(N,KIND=2)` (`N` constante symbolique entière)
  `INDEX(STR,"IDRIS")` ou, `12**4`
  `REAL(I)` n'est pas permis (retourne un réel).
- fonctions intrinsèques d'interrogation :
  `SIZE(A,1)*4` ou, `KIND(0.0D0)`
- certaines fonctions de transformation : `REPEAT,RESHAPE`, `SELECTED_INT_KIND,SELECTED_REAL_KIND,TRANSFER,TRIM`

```
integer,dimension(4,2) :: t = &
        reshape((/(i,i=1,8)/),(/4,2/))
```

**Sont exclues** : `ALL,ANY,COUNT,CSHIFT,DOT_PRODUCT,EOSHIFT, MATMUL,MAXLOC,MAXVAL,MINLOC,MINVAL,PACK,PRODUCT, SPREAD,SUM,TRANSPOSE,UNPACK`.

**Norme 95** : toutes les fonctions intrinsèques sont autorisées ainsi que les fonctions "`pure`" en contexte d'interface explicite. Ces dernières ne doivent alors être ni internes ni récursives et n'avoir aucun argument de type procédural.





# Cours Fortran 95

# Cours Fortran 95
# Annexe F1 : pointeurs et liste chaînée



**Petit exemple de liste chaînée en Fortran 95** :

```
module A
  type cel
    real, dimension(4) :: x
    character(len=10)  :: str
    type(cel), pointer :: p
  end type cel
  type(cel), pointer   :: debut => null()
contains
  subroutine listage
    type(cel), pointer :: ptr_courant
    ptr_courant => debut
    do while(associated(ptr_courant%p))
      print *, ptr_courant%x, ptr_courant%str
      ptr_courant => ptr_courant%p
    end do
    print *, ptr_courant%x, ptr_courant%str
  end subroutine listage
end module A
program liste
  use A
  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 ; nullify(ptr_courant%p)
    ptr_precedent => ptr_courant
    if (ptr_courant%str == "fin") exit
  end do
  call listage
end program liste
```



# Cours Fortran 90/95
# Annexe F2 : pointeurs et COMMON



**COMMON et tableau à profil différé**

```
!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).



# Annexe G : contrôle de procédure F77 via interface "générique et explicite"

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

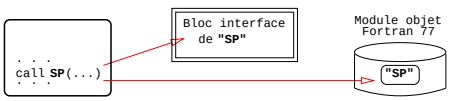



**Schéma 2** : idem en contrôlant le passage d'arguments via un "**bloc interface**".

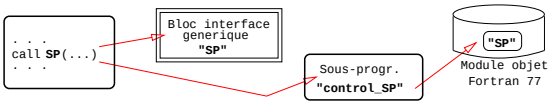



**Schéma 3** : idem en utilisant un **bloc interface générique SP** appelant un sous-programme `control_SP` contrôlant l'appel de `SP` avec la notion d'arguments optionnels et de valeurs par défaut associées (cf. exemple ci-après).



# Annexe G : 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 ! Attention
   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                  !--------------------
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
```

# Annexe H : 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 17)*





## Annexe H1 – Procédures "pure"

Afin de faciliter l'optimisation et la parallélisation des codes, la norme 95 a prévu un nouvel attibut **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.



## Annexe H1 – Procédures "pure"



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).





## Annexe H2 – 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`.

## Annexe H2 – Procédures “elemental” 253

**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
```



## 254 Annexe H3 – 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.



## Annexe H3 – Le “bloc FORALL” 255

**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)
```



## 256 Annexe H3 – Le “bloc FORALL”

**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


– Symboles –

=> ..... 112, 219, 236
=> (use) ..... 171
(/ ... /) ..... 47, 72
(:,... ,) ..... 212
(:,... ,:) ..... 71, 106
.EQ. ..... 29
.GE. ..... 29
.GT. ..... 29
.LE. ..... 29
.LT. ..... 29
.NE. ..... 29
:: ..... 32
; ..... 28
= - pointeurs ..... 115
% ..... 48, 154, 155, 219
& ..... 28

– A –

achar ..... 186
action ..... 180
action - open ..... 179
adjustl ..... 188
adjustr ..... 188
advance - read/write ..... 181, 182
affectation - surcharge ..... 152, 236, 239
alias - pointeurs ..... 110
all ..... 85, 210, 211
allocatable ... 33, 51, 53, 107, 108, 208, 213, 218, 246
allocate .13, 54, 107, 116, 119, 120, 155, 157, 158, 208, 213, 219, 236, 246
allocated ..... 107
anonyme : zone mémoire ..... 237
ANSI ..... 10
any ..... 86, 91, 236
arguments à mot clé ..... 128, 247
arguments optionnels ..... 128, 247
ASA ..... 10
assign ..... 14
associés - pointeurs ..... 110
associated ..... 118, 166, 236
association ..... 235
assumed-shape-array ..... 79
assumed-size-array ..... 77
attributs ..... 33, 140
automatiques - tableaux ..... 106

– B –

bibliographie ..... 21
bloc interface ..... 13, 79, 131, 132, 136, 247
bloc interface nommé ..... 143
bounds checking ..... 18

– C –

case ..... 220
case default ..... 65
case( ) ..... 65
champs - type dérivé ..... 46
char ..... 186
CHARACTER* ..... 16
cible ..... 110, 112
classe ..... 52
commentaires ..... 29, 31
common ..... 124, 136, 165, 246
compatibilité ..... 11
compilation ..... 43
composantes - type dérivé ..... 46
conformance ..... 85, 86
conformants - tableaux ..... 68–70
constructeur - structure ..... 47
constructeur - tableau ..... 47, 72
contains ..... 13, 131, 136
count ..... 87, 211
cpu_time ..... 17, 191
cross-compilateur NEC sxf90 ..... 19, 43
cshift ..... 94, 99, 241
cycle - do/end do ..... 62, 63

– D –

DATA ..... 16
data ..... 12
date ..... 191
date_and_time ..... 191
deallocate ..... 107, 108, 116, 119, 213, 239
déclarations ..... 32, 34, 35
deffered-shape-array ..... 107
delim ..... 180
delim - open ..... 179
dérivée d'une matrice ..... 241
digits ..... 190
dimension ..... 33, 68
do - end do ..... 59, 61
do while ..... 60
documentation ..... 22, 23
dormante : zone mémoire anonyme ..... 237
dot_product ..... 90, 91
dynamiques - tableaux ..... 107

– E –

édition de liens ..... 43
elemental ..... 17, 253
elsewhere ..... 104
end select ..... 65
end= (read) ..... 181, 182
eor= (read) ..... 181, 182
eoshift ..... 96–98
epsilon ..... 190
equivalence ..... 189
états - pointeurs ..... 110, 117
étendue ..... 40, 68, 69, 78
exécution ..... 43
exit ..... 220
exit - do/end do ..... 61–63
exponent ..... 190
external ..... 12, 33

– F –

f90 (IBM) ..... 18, 43, 209
fixed ..... 18
fonction : statement function ..... 16
fonctions élémentaires ..... 103, 104
forall ..... 17, 256
format fixe ..... 16, 31
format libre ..... 16, 28, 30
Fortran 2000 ..... 52
Fortran 95 ....14–17, 21, 46, 53, 82, 84, 104, 108, 110, 117, 145, 165, 166, 169, 183, 184, 191, 193, 243, 249
Fortran Market ..... 23
fraction ..... 190
free ..... 18

– G –

générique : interface ..... 142
GO TO calculé ..... 16

– H –

héritage ..... 52
host association ..... 50, 108
huge ..... 190

– I –

iachar ..... 186
iand ..... 192
ibclr ..... 193
ibits ..... 193
ibset ..... 193
ichar ..... 186
identificateur ..... 27
ieor ..... 192
if - then - else ..... 58
indéfinis - pointeurs ..... 110
index ..... 188
index - fonction ..... 243
initialisation ..... 243
inlining ..... 18
inquire ..... 180
int - fonction ..... 243
intent ..... 13, 33, 128, 135, 154, 158, 221
interface ..... 212
interface assignment ..... 156, 166, 218, 236
interface explicite ..... 79, 106, 128, 129, 131, 132, 136, 139
interface générique ..... 142, 247
interface implicite ..... 126
interface operator ..... 154, 156, 165, 218
internes - procédures ..... 26, 129, 131
intrinsic ..... 33
iolength ..... 180
ior ..... 192
iostat ..... 60, 62, 181, 211
ishft ..... 192
ishftc ..... 192
ISO ..... 11

– K –

kind ..... 13, 36, 38, 196, 197, 212, 243, 253
kind - fonction ..... 39

– L –

lbound ..... 82
len_trim ..... 188, 210, 211
lge ..... 187
lgt ..... 187
liste chaînée ..... 54
lle ..... 187
llt ..... 187
logiques : opérateus ..... 29

– M –

Makefile ..... 209
masque ..... 102
matmul ..... 90, 91
maxloc ..... 82, 84
maxval ..... 88
merge ..... 100
méthodes ..... 160, 165
minloc ..... 82, 84
minval ..... 88
module procedure ..... 143, 154, 156, 165
modules 13, 26, 136, 143, 154, 160, 218, 236, 239
mosaic ..... 23
mots-clé ..... 27, 135
mots-clé - arguments ..... 129
mvbits ..... 193

– N –

namelist ..... 183
nearest ..... 190
NEC SX5 ..... 197
NEC-SX5 ..... 19, 43
netscape ..... 23
normalisation ..... 10
norme ..... 10
not ..... 193
null ..... 17, 110, 117, 165, 239
nullify ..... 117, 120, 166
nuls - pointeurs ..... 110, 116–118

– O –

only (use) ..... 170, 171
open ..... 178
opérateur (nouvel) ..... 154
opérateurs : surcharge ..... 152
opérateurs logiques ..... 29
optional ..... 13, 33, 128, 129, 135, 247

– P –

pack ..... 98, 211, 243
pad ..... 12, 178, 180
parameter ..... 33, 35, 53, 135
pointer ..... 33, 53, 111, 124, 213, 218
pointer - champ de structure ..... 54
pointeurs ..... 54, 110, 235, 246
pointeurs - tableaux ..... 54, 121
polymorphisme ..... 52
position ..... 180
position - open ..... 178
précision ..... 36, 39, 40, 42, 190
present ..... 13, 128, 135, 247
private ..... 33, 160–162, 218, 239
product ..... 89
profil ..... 68, 69, 82, 91, 135
profil différé - tableaux ..... 107
profil différé - tableaux ..... 246
profil implicite - tableaux ..... 79
public ..... 33, 160–162
pure ..... 17, 250, 251

– Q –

-qdbg (IBM) ..... 18
-qfixed (IBM) ..... 18
-qfree (IBM) ..... 18
-qsource (IBM) ..... 18

– R –

ramasse-miettes ..... 237
random_number ..... 191, 207
random_seed ..... 191, 207
rang ..... 68, 69, 72, 246
range ..... 42, 190
real - fonction ..... 243
recl ..... 180
récursif ..... 13
recursive ..... 174
repeat ..... 188
reshape ... 72, 92, 97, 99, 101, 155, 207, 208, 212, 243
result ..... 174
RS/6000 ..... 18, 22, 37, 43

– S –

save ..... 12, 33, 35, 108
scan ..... 188
sections - tableaux ..... 73
sections de tableaux - procédures ..... 80
sections non régulières ..... 76
sections régulières ..... 73
select case ..... 13, 65, 220
selected_int_kind ..... 40, 243
selected_real_kind ..... 40, 243, 253
sequence ..... 50, 53
shape ..... 82, 208, 236
size ..... 78, 82, 106, 208, 212, 243
size (read) ..... 181
sous-types ..... 36, 38
spacing ..... 190
spread ..... 100
stat= ..... 158
static ..... 35
status - open ..... 178
structure ..... 26, 46, 235
sum ..... 89, 91
sur-définis - opérateurs ..... 70
surcharge ..... 13, 139, 237
surcharge d'opérateurs ..... 152
SX5-NEC ..... 19, 43
sxf90 : cross-compilateur NEC ..... 19, 43
system_clock ..... 191

– T –

tableaux automatiques ..... 106
tableaux dynamiques ..... 107, 108
taille ..... 68, 69, 78, 82, 135
taille implicite - tableaux ..... 77
target ..... 53, 108, 112, 124, 218
time ..... 191
tiny ..... 190
traceback ..... 236
transfer ..... 189, 210, 243
transpose ..... 101, 157
tri ..... 122
tridiagonale - matrice ..... 243
trim ..... 188, 243
type ..... 213
type dérivé ..... 46
type dérivé semi-privé ..... 163
types ..... 32

– U –

ubound ..... 82
unpack ..... 99, 243
use ..... 43, 108, 131, 154, 155, 160
use (only) ..... 170
use association ..... 50, 108, 140

– V –

variantes ..... 36
vecteurs d'indices ..... 76
verify ..... 188
vocation - arguments ..... 128

– W –

what ..... 43
where ..... 13, 103, 104, 211, 256
where - elsewhere - end where ..... 102
while - do while ..... 60
WWW ..... 23

– Z –

zone mémoire anonyme dormante ..... 237