Les Evolutions du Fortran 90/95.

B. Nkonga,

Projet ScAlApplix INRIA *Futurs, Université Bordeaux 1*

Plan

- Introduction
- Déclaration des variables
- Les pointeurs
- Les schémas de décision et les schémas itératifs
- Maintenabilité + Réutilisabilité = Modularité
- Les interfaces
- Visibilité
- Vers le Fortran 95/2000/2003

Introduction

Un peu d'histoire

- **1954** IBM avec *John Backus* publie la description d'un système de **FOR**mulation mathématique **TRAN**sposée.
- **1956** Premier manuel de référence qui défini le Fortran I.
- **1957** Fortran II avec l'apparition des procédures et la compilation séparée.
- **1962** Fortran IV (Fortran III est resté interne à IBM) introduit le type explicite. Il sera rebatisé Fortran 66.
- **1978** Fortran 77 (l'ère moderne) propose une avancée significative dans les entrés sorties avec le format libre et l'instruction OPEN.

. . . Un peu d'histoire

- **1991** Fortran 90 quelques facilités de la programmation objets, le calcul matriciel, le contrôle de la précision numérique, ...
- **2000** Fortran 95, introduction de notion de parallélisme de données
- **200X** Fortran 2000, programmation orienté objets

Introduction au génie logiciel (ou rappels)

- La généricité : C'est le fait pour un objet de pouvoir être utilisé tel quel dans différents contextes (ou même indépendamment du contexte).
  - Outils: INTERFACE, MODULE PROCEDURE, OPTIONAL, POINTER
- La modularité : La partie la plus importante de l'écriture de programmes consiste à les structurer pour les présenter comme un assemblage de briques qui s'emboitent naturellement. Ce problème est fondamental dû à la taille conséquente des programmes. La modularité est le fait de structurer un programme en modules indépendants réduisant le coût de la compilation séparée et de la reconstruction incrémentale, tout en maintenant des possibilités d'évolutions.
  - Outils: MODULE, CONTAINS, les types dérivés, structures, dépendances.
- L'encapsulation : L'encapsulation consiste à rassembler des données et/ou des objets au sein d'une structure en masquant l'implémentation de l'objet. Il permet aussi de garantir l'intégrité des données contenues dans l'objet.
  - Outils: PUBLIC, PRIVATE, ONLY, USE, =>

Introduction au génie logiciel

- Le polymorphisme : signifie que les différentes méthodes d'une opération ont la même signature. Lorsque une opération est invoquée sur un objet, celui-ci connait sa classe et par conséquent est capable d'invoquer automatiquement la méthode correspondante. Pour qu'une nouvelle classe supporte une opération existante il lui suffit de fournir la méthode correspondante sans avoir à se soucier des autres méthodes déjà définies.
  - Outils: INTERFACE, "MODULE PROCEDURE", OPTIONAL
- La recurcivité : C'est le fait pour un objet (programme ou une procédure) de s'appeler au moins une fois lui-même. Il permet de résoudre de façon élégante certains problèmes, soit par l'implémentation, soit par le simple fait de penser le problème en terme de récursivité.
  - Outils : RECURSIVE, RESULT, les pointeurs.

Introduction au génie logiciel : le module

Un module est un élément de petite taille (en général un ou quelques sous-programmes) qui sert, par assemblage, à la construction de logiciels. Un module doit être cohérent et autonome. Un module rend des services ou effectue des traitements. Pour exploiter un module dans un logiciel, il est nécessaire d'avoir une description précise de ce qu'il fait, ce qui, dans la pratique se traduit par le passage d'information à travers son interface. De ce point de vue, on peut dire qu'un module est défini par son interface. D'où les principes de la modularité:

- Définir des interfaces explicites chaque fois que deux modules échangent des informations.
- Masquer le plus d'information possible. Seules les informations qui servent à la communication avec d'autres modules doivent être publiques (visibles de l'extérieur du module).
- Un module doit communiquer avec aussi peu d'autres modules que possible.
- Unités linguistiques modulaires : les modules doivent correspondre à des unités syntaxiques du langage.

Structuration d'un programme en F90/95.

La nécessité de décomposer un programme en plusieurs parties est maintenant admise. En Fortran 90/95 chacune de ces parties, appelée unité de programmation, est compilable séparément mais en respectant les dépendances.

- Il existe trois unités de programmation :
  - Le programme principal

```
PROGRAM Toto
CONTAINS
END PROGRAM Toto
```

  - La procédure externe

```
SUBROUTINE Titi(les args)
CONTAINS
END SUBROUTINE Titi
```

```
FUNCTION  Titi(les args)
CONTAINS
END FUNCTION Titi
```

- Le module

```
MODULE Tata
CONTAINS
END MODULE  Tata
```

- Un programme contient au moins l'unité «programme principal».

programme $\neq$ fichier

unité de progammation $\neq$ fichier

Le cadre de programmation : Format libre

- La ligne de programme comporte au plus 132 caractères.
- Les caractères blancs sont significatifs, sauf en début de ligne

```
IF (LaVar.eq.0) THEN et  I F (La Var.eq.0) TH EN
```

- Une ligne de programme peut décrire plusieurs instructions si celles-ci sont séparées par ;

```
a=3 ; b=4
c=8 ; write(*,*) a*b*c
```

- Une instruction peut être décrite sur plusieurs lignes (39 au plus) finissant par "&", sauf la dernière

```
a=38+c* &
& d+b
```

- Dans le passage à la ligne dans l'expression d'une chaîne de caractère la suite de la chaîne doit être précédée par "&"

```
MaString=''bonjour&            MaString=''bonjour&
&    la formation est bien?''  & la formation est bien? ''
```

Le cadre de programmation : Format libre

- le caractère `!` marque le début d'un commentaire qui se termine toujours à la fin de la ligne. Il n'y a pas de restriction sur les caractères utilisés dans un commentaire car ceux-ci sont sans effets. Par conséquent,
  - un commentaire ne peut avoir de suite,
  - un commentaire ne peut être intercalé entre des lignes suites.

```
! on affecte 3 a la variable A
A=3
B=4  ! on affecte 4 a la variable B
```

l'Alphabet

- Les 37 caractères alphanumériques
(26 lettres, 10 chiffres et le souligneur) :

```
a  b  c  d  e  f  g  h  i  j  k  l  m

n  o  p  q  r  s  t  u  v  w  x  y  z

0  1  2  3  4  5  6  7  8  9  _
```

Il y a équivalence entre minuscule et majuscule

- Les 21 caractères spéciaux :

```
+  -  *  /  <  >  .  ,  =  (  )

:  '  !  "  %  &  ;  ?  blanc  $
```

Les éléments lexicaux

Ils sont composés d'au plus 31 caractères alphanumériques dont le premier doit être une lettre.

- Les mots clefs : Ce sont les *noms communs* du langage

```
DO END WHILE MODULE INTEGER MINVAL
```

- Les identificateurs : Ce sont les *noms propres* choisis par le programmeur pour désigner les entités de son programme
  - de variables,
  - de programmes, de procédures, de modules,
  - de types dérivés,
  - d'interfaces génériques, etc.

Les Opérateurs

- Opérateurs arithmétiques

```
+   -   *   /   **      a = b + c*b**2
```

- Opérateurs de comparaison

```
==   /=                 IF (a==3) THEN
```

- Opérateurs d'ordre (uniquement numérique)

```
<   >   <=   >=         IF (a<=3) THEN
```

- Opérateurs logiques

```
.NOT.   .AND.   .OR.   .EQV.   .NEQV.
    IF (.NOT.(a==3).AND.(MyString/=''bonjour'')) THEN
```

- Opérateur de concaténation

```
//              print*, "Bon" // "jour"
```

Séparateurs et délimiteurs

- les séparateurs

```
& , : ; = % :: =>
REAL :: x, Pi, Tab(5)
```

- les délimiteurs

```
/.../ (...) (/.../) ".....".
  Tab(1:5) = (/1, 4, 200, 300, 3/)
```

Les étiquettes

Les étiquettes sont utilisées pour référencer une instruction ou un bloc d'instructions.

- Pour référencer une instruction, l'étiquette est composée d'au plus 5 chiffres située en début de l'instruction.

```
100 FORMAT( E15.8 )
```

- Pour référencer un bloc d'instruction, l'étiquette est composée d'un identificateur en début de bloc suivie du séparateur `:` puis du bloc d'instruction.

```
MaBoucle : DO i = 1, 10
               T(i) = i
           END DO MaBoucle
```

Les instructions exécutables

```
[etiquette] mot-clé [corps]
```

- L'affectation

```
ObjetFinal = expression_evaluable
```

- l'appel à une procédure

```
CALL  MyProcedure(les arguments)
```

- les I/O

```
WRITE(6, *) SinPi(x)
```

- les formats

```
100 FORMAT( E15.8 )
```

- RETURN, STOP

Les “Atomes”

- Les types intrinsèques
  - Numériques : `INTEGER, REAL, COMPLEX`
  - Booléens : `LOGICAL`
  - Chaînes de caractères : `CHARACTER`
- Les opérateurs intrinsèques
  - Scalaires `+, -, *, **, /, ==, >, <, >=, <=,`
  - Booléens `.AND. , .OR. , .NOT.`
  - Chaînes de caractères`//, ==`
  - Affectation =
- Quelques attributs : `PARAMETER, DIMENSION(:,:), ALLOCATABLE, POINTER, TARGET, SAVE, SEQUENCE, INTENT(IN/OUT/INOUT), PUBLIC, PRIVATE, EXTERNAL, INTRINSIC`
- Quelques Mots clefs: `PROGRAM, SUBROUTINE, FUNCTION, MODULE, USE, CALL, CONTAINS, PUBLIC , PRIVATE, PURE, ELEMENTAL, END, DO, ELSE, IF, SELECT, CASE, EXIT, CYCLE, WHILE`

Les “Organes”

- Les fonctions intrinsèques
  - Scalaires: `MAX, MIN, EXP, SIN, COS, MOD`
  - Tableaux:
    `MAX, MIN, EXP, SIN, COS, SUM, MATMUL`
  - Booléens: `PRESENT, ASSOCIATED`
  - Chaînes de caractères: `TRIM, LEN, TRIM_LEN`
- Les procédures utilisateur,
- Les fonctions utilisateur,
- Les modules utilisateur.

Déclaration des variables

F90/95 : Déclaration des variables

```
type [les attributs] ::  liste des objets
 TYPENUM[(paramètres)],          [les attributs] :: liste des vars
 LOGICAL,                        [les attributs] :: liste des vars
 CHARACTER(taille[,paramètres]), [les attributs] :: liste des vars
```

Exemples:

```
INTEGER   , PARAMETER       :: Ns=100, Nt=200
REAL(8)   , PRIVATE         :: pi = 3.14
REAL(4)   , TARGET, SAVE    :: x
REAL(8)   , INTENT(IN)      :: Z
COMPLEX(8), DIMENSION(Ns,3):: ChampsI, ChampsR
LOGICAL   , DIMENSION(Nt)   :: Bol
CHARACTER(LEN=10), PUBLIC   :: VigieName, NetCdfName
CHARACTER(10,KIND(1))       :: VigieName2, NetCdfName2 ! jeu ascii
```

Définitions de quelques attributs

PARAMETER Définit des constantes symboliques

DIMENSION Définit la forme (explicite ou implicite), la taille (explicite, implicite ou différée) et le rang d'un tableau

POINTER Définit les objets dynamiques.

PUBLIC Donne l'accessibilité des composantes d'un Module aux unités qui lui font appel par l'instruction USE.

OPTIONAL permet de définir des arguments optionnels à l'appel d'une procédure.

SAVE Définit des variables rémanentes dans une procédure. La zone mémoire réservée à cette variable est préservée d'un appel à l'autre.

ALLOCATABLE Diffère la réservation de l'espace mémoire à l'utilisation de l'instruction `ALLOCATE`

TARGET Définition des variables cibles potentiels d'objets dynamiques.

SEQUENCE interdit de changer l'ordre des composantes d'une structure.

PRIVATE Limite l'accessibilité des composantes d'un module à lui même.

INTENT Protège l'accès aux arguments d'une procédure en leur donnant une vocation : `IN, OUT, INOUT`.

EXTERNAL Identifie les noms de procédures externes transmis comme arguments à une procédure.

F90/95 : Maîtrise de la précision Numérique

`TYPENUM(KIND=Paramètre) :: les variables`

`TYPENUM` est la notation génériques pour les types intrinsèques numériques : `INTEGER, REAL, COMPLEX.`

- Le paramètre est un entier qui peut s'obtenir en utilisant des fonctions intrinsèques.
  - `SELECTED_INT_KIND(R)` : fonction qui donne la valeur du paramètre du type entier permettant de représenter une entier, X, avec $|X| < 10^r$
  - `SELECTED_REAL_KIND(P,R)` : fonction qui donne la valeur du paramètre du type réel permettant de représenter un réel, X, avec p chiffres significatifs et $|X| < 10^r$
  - `KIND(X)` : fonction qui donne la valeur du paramètre du type X.

F90/95 : Maîtrise de la précision Numérique

- Exemples de déclaration
  - déclaration d'un entier d'au moins 10 chiffres :

```
INTEGER, PARAMETER :: MyIntg0 = SELECTED_INT_KIND(10)
INTEGER(MyIntg0)   :: I
```

  - déclaration d'un réel compris entre -10^{20} et 10^{20} avec 7 chiffres décimaux :

```
INTEGER, PARAMETER :: MyReal0 = SELECTED_REAL_KIND(P=7,R=20)
REAL(MyReal0)      :: X
```

  - déclaration de la précision des réels et des entiers

```
INTEGER, PARAMETER :: MyReal=KIND(1.D0),  MyIntg=KIND(1)
INTEGER(MyIntg)    :: I
REAL(MyReal)       :: X
```

  - déclaration des complexes avec la même précision que pour les réels

```
REAL(KIND=MyReal)  :: X;  COMPLEX(KIND=MyReal)  :: Cx
```

F90/95 : Maîtrise de la précision Numérique

- accès aux limites de précision
  - `EPSILON(X)` écart relatif maximal entre deux réels de type `X`:
    $X(1-\epsilon) > X < X(1+\epsilon)$
  - `HUGE(X)` plus grande valeur représentable dans le type `X`
  - `TINY(X)` plus petite valeur représentable dans le type `X`

```
IInfini = HUGE(1_MyInt);  RInfini = HUGE(1_MyReal)
REps    = EPSILON(1_MyReal)
```

F90/95 : Maîtrise de la précision Numérique

```
PROGRAM PrecisionS
  INTEGER, PARAMETER :: MyReal=KIND(1.0), MyIntg=KIND(1)
  INTEGER, PARAMETER :: MyReal4=KIND(1.E0),MyReal8=KIND(1.D0)
  INTEGER, PARAMETER :: MyRealM=SELECTED_REAL_KIND(19,90)

   PRINT*, " Infini Entier     =", HUGE(1_MyIntg)
   PRINT*, " *****************  REAL par defaut   ******************"
   PRINT*, " Infini Reel       =", HUGE(1.0_MyReal)
   PRINT*, " Eps Relatif Reel  =", EPSILON(1.0_MyReal)
   PRINT*, " Zero   Reel       =", TINY(1.0_MyReal)
   PRINT*, " Precision   10-E  =", PRECISION(1.0_MyReal)
   PRINT*, " *****************  REAL par defaut  1.E0 ******************"
   PRINT*, " Infini Reel       =", HUGE(1.0_MyReal4)
   PRINT*, " Eps Relatif Reel  =", EPSILON(1.0_MyReal4)
   PRINT*, " Zero   Reel       =", TINY(1.0_MyReal4)
   PRINT*, " Precision   10-E  =", PRECISION(1.0_MyReal4)
```

F90/95 : Maîtrise de la précision Numérique

```
  PRINT*, " ****************  REAL par defaut  1.D0 ****************"
  PRINT*, " Infini Reel       =", HUGE(1.0_MyReal8)
  PRINT*, " Eps Relatif Reel  =", EPSILON(1.0_MyReal8)
  PRINT*, " Zero   Reel       =", TINY(1.0_MyReal8)
  PRINT*, " Precision   10-E  =", PRECISION(1.0_MyReal8)
   PRINT*, " ****************  REAL SELECTED_REAL_KIND(6,70)***********
  PRINT*, " Infini Reel       =", HUGE(1.0_MyRealM)
  PRINT*, " Eps Relatif Reel  =", EPSILON(1.0_MyRealM)
  PRINT*, " Zero   Reel       =", TINY(1.0_MyRealM)
  PRINT*, " Precision   10-E  =", PRECISION(1.0_MyRealM)

END PROGRAM PrecisionS
```

F90/95 : Maîtrise de la précision Numérique

Résulats obtenus avec le programme précédent sur un ES45

```
 Infini Entier     =  2147483647
 ****************  REAL par defaut   ******************
 Infini Reel       =  3.4028235E+38
 Eps Relatif Reel  =  1.1920929E-07
 Zero   Reel       =  1.1754944E-38
 Precision   10-E  =           6
 ****************  REAL par defaut  1.E0 ******************
 Infini Reel       =  3.4028235E+38
 Eps Relatif Reel  =  1.1920929E-07
 Zero   Reel       =  1.1754944E-38
 Precision   10-E  =           6
```

F90/95 : Maîtrise de la précision Numérique

```
*****************  REAL par defaut  1.D0 ******************
Infini Reel       =  1.7976931348623l6E+308
Eps Relatif Reel  =  2.220446049250313E-016
Zero   Reel       =  2.2250738585072l0E-308
Precision   10-E  =          15
*****************  REAL SELECTED_REAL_KIND(6,70)************
Infini Reel       =  1.189731495357231765085759326628007E+4932
Eps Relatif Reel  =  1.925929944387235853055977942584927E-0034
Zero   Reel       =  3.362103143112093506262677817321753E-4932
Precision   10-E  =          33
```

Les Tableaux

```
TYPE, DIMENSION(taille1,taille2,...)  :: MonTa
```

- rang limité à 7
- borne inférieure par défaut: 1

```
REAL, DIMENSION(3)    :: MonTab1
REAL, DIMENSION(-1:3) :: MonTab2
REAL, DIMENSION(5:7)  :: MonTab3
```

- Sous tableaux : `Tab(j1:jN:jPas, k1:kN:kPas, ...)`
- les composantes sont rangées en mémoire à des adresses consécutives (colonnes par colonnes).
  Remarque : non garantie pour les sous-tableaux.
- vecteurs anonymes et pseudo-boucles : `(/ /)`

```
REAL, DIMENSION(3)   :: MonTab1=(/0,1,2/)
REAL, DIMENSION(3)   :: MonTab2=(/i**2,i=4,6/)
MonTab2(1)=16  MonTab2(2)=25  MonTab2(3)=36
```

Les “Molécules”: Les Tableaux Statiques

```
type, DIMENSION(10)      :: Tab
type, DIMENSION(Ns,10)   :: Tab1
```

`Ns` est une constante symbolique de type entier.

- Déclaration
  - Les tableaux de type numérique:
    `REAL, DIMENSION(10) :: Tab`
  - De booléens:
    `LOGICAL, DIMENSION(10) :: Tab`
  - De chaînes de caractères:
    `CHARACTER(LEN=5), DIMENSION(10) :: Tab`
- Les éléments d’un tableau sont contiguës en mémoire.

Les Types-dérivés

Ils permettent de définir des structures de données complexes

- Définition d'un type-dérivé.

```
TYPE [Attribut du TYPE ::] Nom_du_type
  [Attribut des composants]
  type [les attributs des champs] :: les cha
END TYPE [Nom_du_type]
```

- Les attributs du type : `PUBLIC, PRIVATE`.
- Les attributs des composantes : `SEQUENCE, PUBLIC, PRIVATE`.
- Les attributs **proscrits** pour les champs sont:

```
PARAMETER, 'ALLOCATABLE', TARGET, SAVE, OPTIONAL, 'PRIVATE', 'PUBLIC'
```

Les Types dérivés privés

- La différence entre ces deux déclarations.
  - On n'accède pas au type en dehors de la portée

```
TYPE, PRIVATE :: POINT3D
  REAL(8), DIMENSION(3) :: pos
END TYPE POINT3D
```

  - on n'accède pas aux composants du type en dehors de la portée. On parle alors de type abstrait.

```
TYPE POINT3D
  PRIVATE
  REAL(8), DIMENSION(3) :: pos
END TYPE POINT3D
```

Types dérivés : exemples

```
! Les sommets en 2D

TYPE Point
     REAL(8) :: X,Y
END TYPE Point
```

```
!La connectivité d'un triangle

TYPE Triangle
     INTEGER(KIND(1)) :: S1,S2,S3
END TYPE Triangle
```

```
! Le maillage géométrique

TYPE Maillage2D
 Point,    DIMENSION(200) :: Noeuds
 Triangle, DIMENSION(150) :: Connectivite
END TYPE Maillage2D
```

Les Types-dérivés statiques

Exemples

```
TYPE Maillage
 INTEGER(KIND=MyIntg)                    :: Ns=10, Nt=20
 INTEGER(KIND=MyIntg), DIMENSION(10)  :: LogN
 INTEGER(KIND=MyIntg), DIMENSION(3,20):: Nu
 REAL(KIND=MyReal)   , DIMENSION(2,20):: Coor
 COMPLEX(KIND=MyReal), DIMENSION(2,20):: Z
END TYPE Maillage


TYPE LesChamps
 COMPLEX(KIND=MyReal), DIMENSION(3,200):: Ex
 COMPLEX(KIND=MyReal), DIMENSION(3,200):: Ey
 COMPLEX(KIND=MyReal), DIMENSION(3,200):: Ez
END TYPE LesChamps
```

Les “Molécules”: Les Structures Statiques

- Création de structures statiques:

```
TYPE(Maillage)           :: Niveau0
TYPE(Maillage), PRIVATE :: NiveauL
```

- Les Tableaux statiques de structures statiques:

```
TYPE(Maillage), DIMENSION(Ns)     :: Hyb
TYPE(Maillage), DIMENSION(10,100) :: HybS
```

- Accès aux composantes: le sélecteur “%“

```
Niveau0%Coor(1,2)   = 0.7_MyReal
HybS(1,2)%Coor(1,2) = 0.3_Myreal
Niveau0%Z(1,2)      = (0._MyReal, 7._MyReal)
```

Les “Cellules” : Objets Dynamiques

- Objets à allocation différée

```
type, ALLOCATABLE                    :: ScaDyn
type, ALLOCATABLE, DIMENSION(:)      :: TabD
type, ALLOCATABLE, DIMENSION(:,:)    :: TabDyn
```

La réservation de la mémoire n’est effective qu’avec l’instruction `ALLOCATE` et la libération de cet espace qu’avec l’instruction `DEALLOCATE`

```
ALLOCATE( ScaDyn, TabD(1:200), TabDyn(-1:200,0:200) )
TabD=(/ (2.0*i, i = 0, 199) /)
DEALLOCATE( TabD )
...
READ*, n
ALLOCATE( TabD(1:n), stat = err )
if( err /=0 ) THEN
    TabD=(/ (2.0*i, i = 0, n) /)
ENDIF
```

Les “Cellules” : Objets Dynamiques

- Tableaux à dimension automatique, pour des arguments de procédures:

```
 type, DIMENSION(:,:)    , INTENT(IN) :: TabDyn
 type, DIMENSION(-1:,0:) , INTENT(IN) :: TabDyn
 type, DIMENSION(-1:,-2:), INTENT(IN) :: TabDyn

Nx=SIZE(TabDyn, DIM=1) ; Ny=SIZE(TabDyn, DIM=2)
i0=LBOUND(TabDyn,DIM=1); j0=LBOUND(TabDyn,DIM=2)
iN=UBOUND(TabDyn,DIM=1); jN=UBOUND(TabDyn,DIM=2)
-------------------------------------------------------
a) Nx=202 Ny=201 i0=1  j0=1  iN=202 jN=201
b) Nx=202 Ny=201 i0=-1 j0=0  iN=200 jN=200
c) Nx=202 Ny=201 i0=-1 j0=-2 iN=200 jN=198
```

- Tableaux (TabLoc) locaux à une procédure, de taille dépendant de celle d’un argument de la procédure.

```
 type, DIMENSION(-1:,0:) , INTENT(IN) :: TabDyn
 type, DIMENSION(SIZE(TabDyn,DIM=1))  :: TabLoc
```

Les pointeurs

Les Pointeurs

- Création d'un pointeur

```
type, POINTER [ autres attributs] :: Alias
type, POINTER [ autres attributs] :: Alias=>NULL()
```

  dans le premier cas `Alias` est dans un état indéterminé, dans le second cas il est dans l'état libre.

- Assignation d'un pointeur libre : `Alias => Cible`
  `Cible` est un objet du même type et de même forme que `Alias`, ayant l'attribut `TARGET` ou `POINTER`.
- Rompre toute assignation du pointeur: `NULLIFY(Alias)`.
- Création d'une cible anonyme et assignation sur cette cible : `ALLOCATE(Alias)`.
  Pour rompre l'association dans ce cas on utilise l'instruction `DEALLOCATE`
- Etat d'association d'un pointeur : `ASSOCIATED(POINTER=Alias, TARGET=Cible)`, retourne `.TRUE.` si `Alias` est associé avec `Cible`, `.FALSE.` sinon. L'argument `TARGET` est optionnel: `ASSOCIATED(POINTER=Alias)`, retourne `.TRUE.` si `Alias` est associé et `.FALSE.` sinon.

Pointeurs et passage en argument

- Pointeur passé en argument d'appel d'une procédure.
  - 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.
  - L'argument muet a l'attribut pointer
    - le pointeur n'est pas nécessairement associé avant l'appel (avantage par rapport à allocatable),
    - c'est l'adresse du descripteur du pointeur qui est passée,
    - l'interface doit être explicite (pour que le compilateur sache que l'argument muet a l'attribut pointer),
    - si le pointeur passé est associé à un tableau avant l'appel, les bornes inférieures/supérieures de chacune de ses dimensions sont transmises à la procédure ; elles peuvent alors être récupérées via les fonctions UBOUND/LBOUND.

Pointeurs et passage en argument

- 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.
  - 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.

Procédures recursives

- La récursivité permet de définir une procédure ou une fonction capable de s'appeler elle-même.
  - Définition d'une procédure récurcive:

```
RECURSIVE SUBROUTINE TriPartition( )
    ....
END SUBROUTINE TriPartition
```

  - Définition d'une fonction récurcive

```
RECURSIVE FUNCTION Factoriel(n ) RESULT(m)
    ....
END FUNCTION Factoriel
```

Procédures recursives: Exemple

```
RECURSIVE FUNCTION Factoriel(n )RESULT(m)

  INTEGER, INTENT(IN) :: n

  INTEGER             :: m
  ! ------------------------------------

  IF( n == 0 ) THEN
     m=1
  ELSE
     m=n*Factoriel(n-1)
  END IF

END FUNCTION Factoriel
```

Structures dynamiques

- Définition d'un type-dérivé ayant des composantes dynamiques:

```
TYPE MaillageD
  INTEGER(KIND=MyIntg)      :: Ns, Nt, NsFr=0
  INTEGER(KIND=MyIntg), DIMENSION(:)  , POINTER :: LogN
  INTEGER(KIND=MyIntg), DIMENSION(:)  , POINTER :: LogFr
  INTEGER(KIND=MyIntg), DIMENSION(:,:), POINTER :: Nu
  INTEGER(KIND=MyIntg), DIMENSION(:,:), POINTER :: NuFr
  REAL(KIND=MyReal)   , DIMENSION(:,:), POINTER :: Coor
END TYPE MaillageD
```

- Définition de structures dynamiques:

```
TYPE(MaillageD) :: Niveau0
TYPE(MaillageD) , DIMENSION(:), ALLOCATABLE :: HybD1
TYPE(MaillageD), DIMENSION(:), ALLOCATABLE :: HybDD
```

Listes chaînées

- type-dérivé de la cellule élémentaire

```
TYPE Cell
  (composantes)
  TYPE(Cell), POINTER :: Suiv=>NULL()
END TYPE Cell
```

- Tête de liste `TYPE(Cell) :: Debut`
- Variable intermédiaire : `TYPE(Cell), POINTER :: TheCell`
- Création de la liste

```
ALLOCATE( TheCell ) ; Debut%Suiv=> TheCell
ALLOCATE( TheCell )
Debut%Suiv%Suiv => TheCell
```

Les schémas de décision
Les schémas itératifs

L'alternative: bloc IF

- Réalisation d'instructions sous condition

```
[nom:] IF( test_1 ) THEN
          instructions
       ELSE IF(test_2) THEN
          instructions
       ELSE
          instructions
       END IF [nom]
```

```
IF( a == 3 ) THEN
          WRITE(*,*) 'a vaut 3'
       ELSE IF( a>3 ) THEN
          WRITE(*,*) 'a plus grand que 3'
       ELSE
          WRITE(*,*) 'a plus petit que 3'
END IF
```

- Les blocs IF peuvent être emboités
- Bloc IF sur une ligne :

```
IF( test_1 ) instruction
```

```
IF ( A < 0 )  A = -A
```

l'aiguillage multiple: bloc SELECT

- Selection d'une séquence d'instruction parmi N séquences possibles

```
nom: SELECT CASE (critère)
      CASE(choix1)
       Instructions
      CASE(choix2)
       Instructions
     CASE DEFAULT
          Les autres cas
     END SELECT nom
```

```
SELECT CASE(meth)
CASE('milieu','cen')
  call milieu
CASE('trapeze')
  call trapeze
CASE DEFAULT
  call gauche
END SELECT
```

- Le critère est soit un entier soit une chaîne de caractère

```
choix = i1:iN,  2,     9,       j1:,  :JN
choix = "Lala", "Pz", 'a':'z'
```

bloc DO

- Répéter une séquence d'instructions suivant un compteur
  Utilisation :

```
[nomD :] DO indice= debut, fin, pas
            Instructions
         END DO [nomD]
```

- Si pas est négatif le comptage est en sens décroissant
- Exemples

```
Intg : DO  jt =1, Nt, 2
           Sum = Sum + aire(jt)
        END DO Intg
```

- Boucle infinie

```
DO
   i = i+1
   IF (i>=10) STOP
END DO
```

Les Boucles : DO WHILE

- Faire une séquence d'instructions *tant que* une condition est réalisée
- Utilisation :

```
[nomW :] DO  WHILE (test)
        Instructions
        modification du test
      END DO [nomW]
```

- Exemple

```
cas : DO  WHILE (x < 5)
          s = s + x
          x = x + 1
    END DO cas
```

Alternative vectorielle : WHERE

Affectation conditionnelle dans un tableau

```
  WHERE (test)
    Instructions
  [ ELSEWHERE (test2)
       Instructions ]
  [ ELSEWHERE
       Instructions ]
 END WHERE
```

```
WHERE( Rho> 0 )
  Tab2=1.0/Rho
ELSEWHERE(Pes >0)
  Tab2=0.
ELSEWHERE
  Tab2=HUGE(1.0)/10
END WHERE
```

`test` et `test2` sont des tableaux de booléens ayant les mêmes caractéristiques.

Les Boucles: CYCLE et EXIT

- `CYCLE [IdBoucle]` : il permet d'interrompre l'itération en cours et de passer à la suivante. Lorsqu'il y a plusieurs boucles imbriquées, elle permet d'interrompre l'itération la boucle étiquetée par `IdBoucle` si cet argument est spécifié, sinon la boucle interrompue est celle dans laquelle on se trouve.
- `EXIT [IdBoucle]` : il permet de sortir de la boucle en cours, avec les mêmes options que précédemment.

```
Iter : DO kt=1,KtM
          DO i = 1, Ns
             X(i) = (i-1)*Dx
             IF( X(i) > 1000.0 ) EXIT Iter
             IF( X(i) > 10.0 ) CYCLE
             TAB(i) = X(i)*X(i) - 1.0
          END DO
       END DO Iter
```

```
cas : DO
         IF  (x < 5) EXIT
         s = s + x
         x = x + 1
      END DO cas
```

Maintenabilité

+

Réutilisabilité

=

Modularité

Quelques règles de programmation

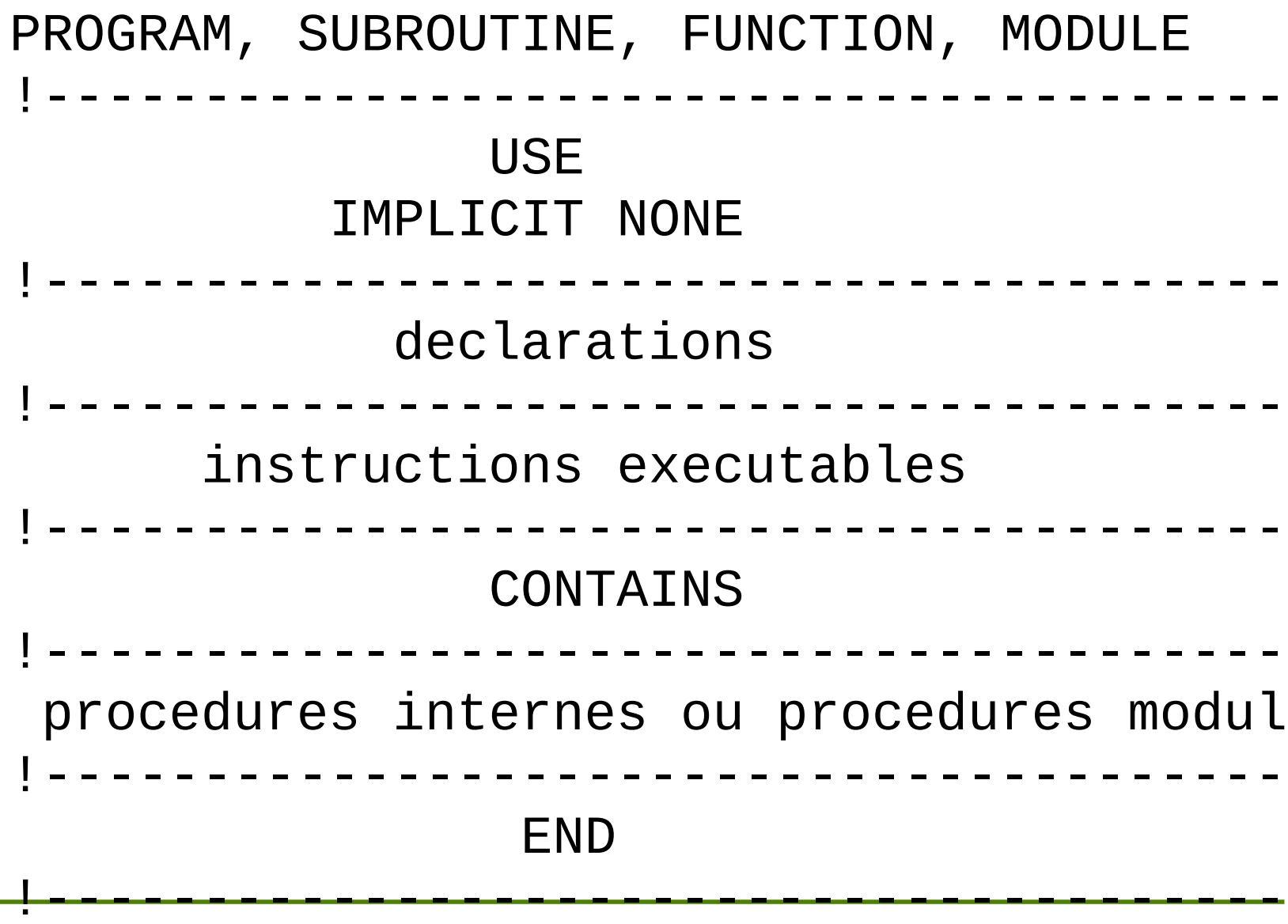
- Utiliser le format libre
- Le `COMMON` est à proscrire, utiliser un module
- Ne pas utiliser la déclaration de type implicite. Tous les programmes, les modules, les subroutines et les fonctions doivent **obligatoirement** commencer par la spécification

```
IMPLICITE NONE
```

“La Bète”: Le programme principal

```
PROGRAM MonProgramme
   IMPLICIT NONE

   [ specification et declarations ]

   !**********************************************

   [instructions executables]

[CONTAINS

         procedures internes ]

END  PROGRAM Monprogramme
```

Ordre des instructions



La Procédure

- Une procédure définit une fonction (`FUNCTION`) ou un sous-programme (`SUBROUTINE`). Elle pourra être:
  - une unité de programmation F90. Il s'agit d'une procédure externe.
  - construite à partir d'autres langages que le Fortran.
  - une composante d'une unité de programmation module. Il s'agit d'une procédure module.
  - à l'intérieur d'une procédure module, d'une procédure externe ou d'un programme principal. Il s'agit d'une procédure interne.
- Une unité de programmation contenant une procédure interne ou une procédure module est appelée unité hôte.

L'unité de programmation Procédure

Une action

```
SUBROUTINE fft(s,m)
  IMPLICIT NONE

  [ specifications
  et declarations ]

  [instructions
   executables]

[CONTAINS
 procedures
    internes ]
END SUBROUTINE fft
```

Un calcul

```
FUNCTION fftM(x) RESULT Res
  IMPLICIT NONE

  [ specifications
  et declarations ]

  [instructions
   executables]

[CONTAINS
 procedures
    internes ]
END FUNCTION fftM
```

Procédure : Arguments

- l'attribut INTENT permet de spécifier la vocation des arguments d'une procédure sauf si ce sont des pointeurs ou des procédures.
  - argument d'entré : `INTENT(IN)`
  - argument de sortie : `INTENT(OUT)`
  - argument mixte : `INTENT(INOUT)`
- L'attribut OPTIONAL permet de spécifier des arguments dont la présence n'est pas obligatoire à l'appel de la procédure.

  ```
  REAL(KIND=8), OPTIONAL :: X2, X4
  ```

  La fonction intrinsèque `PRESENT` permet d'enquêter sur la présence ou non d'un argument optionnel à l'appel de la procédure.

  ```
  IF(PRESENT(X4) ) X=2*X4
  ```
- Le passage par mot-clé est fortement conseillé s'il y a des arguments optionnels, de plus il permet d'oublier la position des arguments lors de l'appel de la fonction;

Procédure : Arguments

- les procédures en argument doivent être déclarées avec l'attribut `EXTERNAL`.
  - Les procédures internes ne peuvent être passées en argument.
  - Pour les procédures externes, il est recommandé de fournir une interface de description à l'unité appelante.
  - C'est le nom spécifique de la procédure qui doit être fourni même si elle a un nom générique.

```
REAL(4), EXTERNAL :: SinPi
EXTERNAL   :: SinPiS
```

- La valeur de retour d'une fonction doit être déclaré

```
FUNCTION test (X)
REAL :: test

REAL FUNCTION test (X)
```

```
FUNCTION test (X) RESULT res
REAL :: res
```

Le rôle du MODULE

- Un module permet de structurer le programme en fonction de thématiques: Résolution de systèmes linéaires et non linéaires, Solveurs de Riemann, Communications MPI ou PVM, Entrés/Sorties, Définitions et données partagées.
- Un Module peut contenir (composantes):
  - des spécifications `USE, IMPLICIT NONE`
  - des déclarations d'objets `REAL :: Pi = 3.14`
  - des définitions de types dérivés,
  - des blocs-interfaces (avant le `CONTAINS`)
  - un ensemble de procédures (après le `CONTAINS`).
- Ces composantes sont empaquetées sous une forme qui les rend accessibles n'importe où dans le programme.

Création d'un MODULE

```
MODULE Meth_Relaxation
   IMPLICIT NONE
   [ specification et declarations
   d'objets globaux   au module  ]

[CONTAINS ! composantes procedures modules
    SUBROUTINE GMRESM(A, B, m, X)
    [ specification et declarations ]

    [instructions executables]
    [CONTAINS
         procedures internes ]
     END   SUBROUTINE GMRESM ]
END MODULE Meth_Relaxation
```

Accessibilité d'un MODULE

- Par défaut, toutes les ressources d'un module sont accessibles.
- Il peut être souhaitable de limiter l'accessibilité de ces ressources. Ceci peut se justifier lorsqu'elles ne sont nécessaires qu'à l'intérieur du module dans lequel elles sont définies.
- Les ressources non exportables sont dites privées (`PRIVATE`), les autres sont dites publiques (`PUBLIC`).
- Les avantages des ressources privées sont :
  - Aucune corruption accidentelle sur des données privées par une procédure externe au module qui les contient,
  - des modifications de conceptions peuvent être faites sur les ressources privées d'un module sans affecter le reste du programme,
  - permet d'éviter tout type de conflits avec les ressources d'autres modules...

Accessibilité d'une composante d'un MODULE

- La composante d'un module peut être
  - Publique: c'est à dire accessible à partir des autres composantes du module et des unités de programmation qui utilisent le module.

```
REAL(8), PUBLIC  :: Pi
REAL(4), PUBLIC  :: Hbar
PUBLIC           :: GMRESM
```

  - Privée: c'est à dire accessible uniquement à partir des autres composantes du module en question.

```
REAL(8), PRIVATE  :: Pi
PRIVATE           :: GMRESM
```

- Tous les objets déclarés dans un MODULE sont rémanents (F95) et globaux pour les procédures modules.

Accessibilité des composantes d'un MODULE

- Par défaut, toutes les composantes d'un module sont publiques.
- Quand la pseudo-instruction `PRIVATE` apparaît toute seule dans la partie déclaration du module, toutes les composantes deviennent par défaut privées.

```
MODULE Definitions
  IMPLICIT NONE
  PRIVATE

  CONTAINS

END MODULE Definitions
```

Le Module: USE, ONLY, =>

- Les composantes publiques d'un module sont accessibles depuis une autre unité de programmation par l'instruction USE située en tête des déclarations et spécifications.

```
USE Definitions
```

- Dans une unité de programmation, on peut restreindre l'accès à quelques composantes publiques d'un module par l'instruction ONLY à la suite de USE.

```
USE Definitions, ONLY : Pi
USE Definitions, ONLY : Pi, SinPi
USE Definitions, ONLY :          ! rien d'utile
```

- En cas de conflit avec un identificateur dans une unité de programmation, on peut changer localement le nom d'accès à des composantes publiques d'un module par l'utilisation du pointeur => à la suite de USE.

```
USE Definitions, MyPi => Pi              ! MyPi est un alias de Pi
USE Definitions, ONLY : MyPi=>Pi, SPi=>SinPi
```

Les interfaces

La fiabilisation des appels d’une procédure

Les types d'interfaces

- Interface simple ou de description : Permet de décrire les arguments d'appel d'une procédure ainsi que leur vocation (arguments d'entrée ou de sortie).
- Interface générique : Permet de définir un nom générique pour l'appel de plusieurs procédures de même nature (`subroutine` ou `function`). Le choix de la procédure executée lors de l'appel dépend des arguments.
- Interfaces opérateurs : Permet de définir de nouvelles opérations entre des types intrinseques ou non. Elle permet également d'étendre des opérateurs intrinsèques; on parle alors de surcharge.
- Interfaces d'affectation : Permet d'étendre l'opérateur d'affection (=) à des types non-intrinseques.

Interface simple ou de description

```
INTERFACE
 SUBROUTINE Tata(Champs,Surf,SER)
   TYPE(LesChamps)  , INTENT(IN):: Champs
   TYPE(LesSurfaces), INTENT(IN):: Surf
   REAL(8)                      :: SER
 END SUBROUTINE Tata
END INTERFACE
```

F90/95 : Interface "explicite" automatique

- Les procédures intrinsèques;
- Les procédures internes;
- Les procédures modules;

la procédure appelante accède au module contenant le bloc d'interface de la procédure appelée (USE).

F90/95 : Interface "explicite" obligatoire

- fonction à valeur tableau,

```
INTERFACE
 FUNCTION Tata(Champs) RESULT MonTab
   TYPE(LesChamps)  , INTENT(IN):: Champs
   REAL(8), DIMENSION(12)       :: MonTab
 END FUNCTION Tata
END INTERFACE
```

- fonction à valeur pointeur,
- fonction à valeur chaîne de caractères dont la longueur est déterminée dynamiquement,

```
INTERFACE
 SUBROUTINE Tata(car1,car2)
   CHARACTER(*), INTENT(IN)   :: car1
   CHARACTER(*), INTENT(OUT)  :: car2
 END SUBROUTINE Tata
END INTERFACE
```

F90/95 : Interface "explicite" obligatoire

- tableau à profil implicite,

```
INTERFACE
 SUBROUTINE Tata(MonTab)
   REAL, DIMENSION(:,:)    :: MonTab
 END SUBROUTINE Tata
END INTERFACE
```

- argument formel avec l'attribut pointer ou target,
- passage d'arguments à mots-clé,
- argument optionnel,

```
INTERFACE
 SUBROUTINE Tata(MonTab, X)
   REAL, DIMENSION(:,:)    :: MonTab
   REAL, OPTIONAL          :: X
 END SUBROUTINE Tata
END INTERFACE
```

Mots-clé : un exemple

```
INTERFACE
  SUBROUTINE test(X1,X2,X3,X4)
    REAL(KIND=8), INTENT(IN), OPTIONAL :: X2, X3,X4
    REAL(KIND=8), INTENT(INOUT) :: X1
  END SUBROUTINE test
END INTERFACE
```

Les différent appels

```
 call test(a,b,c,d)              ! tous les arguments sont présents

 call test(a)                    ! pas d'arguments optionnels

 call test(a,X4=d)               ! X2,X3 sont absents

 call test(a,b,X4=d)             ! X3 est absent
 call test(a,X2=b,X4=d)          ! la forme à utiliser

 call test(X4=d,X1=a,X2=b,X3=c) ! passage par mot clé
```

F90/95 : Interface "explicite" obligatoire

- procédure générique,
- surcharge ou définition d’un opérateur,
- surcharge de l’opérateur d’affectation.

Interfaces génériques

L'Interface générique donne le cadre pour invoquer une famille de procédures distinctes au moyen d'un même nom: "nom générique". Le choix de la procédure à exécuter est déterminé en fonction du nombre et du type des arguments.

```
INTERFACE MySend
 SUBROUTINE Tata(Champs,Surf,SER)
   TYPE(LesChamps)  , INTENT(IN):: Champs
   TYPE(LesSurfaces), INTENT(IN):: Surf
   REAL(8)                      :: SER
 END SUBROUTINE Tata

 SUBROUTINE TataN(Champs,Surf,SER)
   TYPE(LesChamps)  , INTENT(IN):: Champs
   TYPE(LesSurfaces), INTENT(IN):: Surf
   REAL(4)                      :: SER
 END SUBROUTINE TataN
END INTERFACE
```

Interfaces génériques

```
INTERFACE MySend
 SUBROUTINE Titi_1(Champs,Surf,SER)
   TYPE(LesChamps)  , INTENT(IN):: Champs
   TYPE(LesSurfaces), INTENT(IN):: Surf
   REAL(8)                      :: SER
 END SUBROUTINE Titi_1

 SUBROUTINE Titi_2(Champs,SER,RES,N)
   TYPE(LesChamps)  , INTENT(IN):: Champs
   REAL(4)                      :: SER, RES
   INTEGER                      :: N
 END SUBROUTINE Titi_2

 SUBROUTINE Titi_3(Champs,Surf)
   TYPE(LesChamps)                  , INTENT(IN):: Champs
   TYPE(LesSurfaces), DIMENSION(4), INTENT(IN):: Surf
 END SUBROUTINE Titi_3
END INTERFACE
```

Interfaces : Nouveaux opérateurs

- Pour définir un nouvel opérateur, on utilise l'interface operator.

```
INTERFACE OPERATOR( .op. )
  MODULE PROCEDURE ComplxMult
END INTERFACE
```

`.op.` est un nouvel opérateur et `ComplxMult` est la (ou les) fonction définissant l'opérateur.

- L'appel de ce nouvel opérateur ce fait de la façon suivante :

```
A = B .op. C
```

Interfaces : Surcharge d'opérateurs

- La surcharge d'opérateurs intrinsèques du langage permet d'élargir leur champ d'application. A condition de respecter la nature (binaire ou unaire) et les règles de priorité définies par le langage. On emploie des procédures de type fonction pour surcharger un opérateur à l'exception de l'opérateur d'affectation qui construit une expression ne retournant aucune valeur. Dans ce cas, c'est une procédure de type subroutine qui sera utilisée.
- Pour surcharger un opérateur (autre que l'opérateur d'affectation), on utilise l'interface operator.

```
INTERFACE OPERATOR( op )
  MODULE PROCEDURE ComplxMult
END INTERFACE
```

op est l'opérateur intrinsèque que l'on désire surcharger et `ComplxMult` est la (ou les) fonction définissant la surcharge.

Interfaces : Surcharge d'opérateur

En général, on définira la surcharge d'opérateur dans un module.
Lors de la surcharge d'un opérateur autre que l'opérateur d'affectation, le ou les arguments de la fonction associée doivent avoir l'attribut intent(in).

```
MODULE MesTypes
  IMPLICIT NONE
   TYPE MyComplex
     REAL  :: Preel, PImag
  END TYPE MyComplex
END MODULE MesTypes
```

```
MODULE NewOp
  USE MesTypes
  !
  INTERFACE OPERATOR(*)
     MODULE PROCEDURE ComplxMult
  END INTERFACE
  INTERFACE OPERATOR(/)
     MODULE PROCEDURE ComplxDiv
  END INTERFACE
                          .......
```

F90/95 : Surcharge d'opérateur

```
CONTAINS

  FUNCTION ComplxMult(C1, C2)
    TYPE(MyComplex), INTENT(in)   :: C1, C2
    TYPE(MyComplex)               :: ComplxMult
    !
    ComplxMult%Preel = C1%Preel*C2%Preel - C1%Pimag*C2%Pimag
    ComplxMult%Pimag = C1%Preel*C2%Pimag + C1%Pimag*C2%Preel
  END FUNCTION ComplxMult

  FUNCTION ComplxDiv(C1, C2)
    TYPE(MyComplex), INTENT(in)   :: C1, C2
    TYPE(MyComplex)               :: ComplxDiv
    !
    ComplxDiv%Preel = C1%Preel*C2%Preel + C1%Pimag*C2%Pimag
    ComplxDiv%Pimag = C1%Preel*C2%Pimag - C1%Pimag*C2%Preel
  END FUNCTION ComplxDiv
END MODULE NewOp
```

F90/95 : Surcharge d'opérateur

```
PROGRAM Test
  USE NewOp
  TYPE(MyComplex) :: A1, A2

  A1=MyComplex(2,1) ; A2=MyComplex(3,4)

  print*, " A1= ", A1 ; print*, " A2= ", A2
  print*, " A1*A2= ", A1*A2
  print*, " A1/A2= ", A1/A2
END PROGRAM Test
```

F90/95 : Surcharge d'opérateur ASSIGNMENT

Pour surcharger l'opérateur assignment (=), on utilisera un bloc interface du type interface assignment.
La définition de l'opérateur "=" se fera dans un module. Le sous-programme qui définit l'assignation pourra être un sous-programme externe ou interne au module. On préférera cette dernière méthode.

Lors de la surcharge de l'opérateur d'affectation, le 1er argument doit avoir l' attribut intent(out) ou intent(inout) et le 2eme l'attribut intent(in).

F90/95 : Surcharge d'opérateur ASSIGNMENT

Exemple

```
MODULE Assign
  INTERFACE ASSIGNMENT(=)
     MODULE PROCEDURE Char2Int
  END INTERFACE

CONTAINS

  SUBROUTINE Char2Int(n,c)
    INTEGER,       INTENT(OUT) ::n
    CHARACTER(*), INTENT(IN)  ::c
    ........
  END SUBROUTINE

END MODULE
```

Visibilité

Visibilité

- Les attributs PUBLIC et PRIVATE ne peuvent apparaître qu'à l'intérieur d'un module.
- L'instruction PUBLIC ou PRIVATE sans argument ne peut apparaître qu'une seule fois dans un module,
- Si une procédure a un identificateur générique, l'accessibilité à son nom spécifique est indépendante de l'accessibilité à son nom générique,
- Une entité dont le type a été défini avec l'attribut PRIVATE ne peut pas posséder l'attribut PUBLIC,
- si une procédure a un argument formel ou un résultat avec un attribut PRIVATE, la procédure doit être munie de l'attribut PRIVATE.

F90/95 : Visibilité

- Variables et objets publiques ou privés:

```
REAL                , PUBLIC          :: Coor
INTEGER             , PRIVATE         :: VigieName
TYPE(MyComplex), PRIVATE              :: C1, C2
```

- Modification du mode par défaut

```
MODULE Tata ; PRIVATE
```

- liste d'objets de variables ou de procédures non affectés par le mode par défaut:

```
PRIVATE ComplxMult, RCMult1, RCMult2
PUBLIC  ComplxDiv, ComplxEq
```

Visibilité d'un type dérivé

Les attributs PUBLIC et PRIVATE peuvent s'appliquer aux types dérivés.
Un type dérivé peut être:

- public ainsi que ses composantes: type dérivé transparent,
- privé,
- public mais avec toutes ses composantes privées: type dérivé semi-privé.
  Par défaut les composantes d'un type dérivé public sont publiques.

Visibilité d'un type dérivé

L'utilisation d'un type dérivé semi-privé présente l'avantage de permettre des changements sur le type sans affecter de quelque manière que ce soit les unités utilisatrices. Reprenons l'exemple du type "privatecomplex". On transforme le type public en type semi-privé.

```
type private_complex
   private
 real   :: reel, im
end type private_complex
```

Les composantes de ce type étant privées, il est nécessaire de fournir dans le module qui le définit des fonctions qui permettent d'y accéder ou de les modifier.

F90/95 : Visibilité

Lors de l'utilisation d'un module, il se peut que dans l'unité utilisatrice, il existe des ressources ayant le même nom. Dans ce cas,il est possible de renommer les ressources du module dans l'unité qui y accède via l'opérateur "" au niveau de l'instruction

```
USE NewOp, NewName => ComplxMultP
```

Lorsque seulement un sous-ensemble des noms définis dans un module est requis, l'option ONLY est disponible, avec la syntaxe suivante:

```
USE NewOp, ONLY: NewName => ComplxMultP
```

Vers le Fortran 95/2000/2003

Séquences parallèles : FORALL

Contrôler l'exécution d'instructions (affectation, association, ...) afin de faciliter leurs distributions sur les processeurs

```
! Utilisation
[nom:]  FORALL ( indices, indices [, filtre] )
   Instructions
END FORALL [nom]

!Exemple

FORALL( i = 1:100:5, j = 1:10 , i > j)
  Tab2(i) = i
  Tab2(j) = j
END FORALL
```

Procédure sans effet de bord : PURE

Une fonction avec l'attribut PURE est une fonction sans effet de bord

```
! Utilisation

PURE FUNCTION ThePure( les, arguments )

END FUNCTION ThePure
```

- Ses paramètres autres que les pointeurs et les procédures doivent avoir l'attribut `INTENT(IN)`.
- L'attribut SAVE est interdit, on ne doit pas modifier une variable globale.
- Pas d'ordre STOP ni des I/O dans les instructions.
- on peut appeler la fonction dans une boucle FORALL.

Procédure distributive : ELEMENTAL

Une fonction (ou procédure) distributive est une fonction pure qui n'admet que des paramètres scalaires, autres que les pointeurs.
Le résultat doit être un scalaire.

```
! Utilisation

ELEMENTAL SUBROUTINE PERMUT(X,Y)
   REAL(8), intent(inout) :: X,Y;
   REAL                   :: Temp

   Temp = X ; X = Y ; Y = Temp

END SUBROUTINE PERMUT
```

F90/95 : Caractéristiques obsolètes

On désigne par caractéristiques obsolètes, les caractéristiques qui sont susceptibles d'être éliminées lors d'une prochaine phase de normalisation. Ce sont des caractéristiques redondantes et pour lesquelles une meilleure méthode peut être utilisée.

- `IF` arithmétique.
- Branchement sur une instruction `END IF` depuis l'extérieur du bloc `IF` - `END IF`.
- Variation de la syntaxe d'une boucle `DO` * Partage d'une même instruction par plusieurs boucles `DO`. * Fin de boucle `DO` autres que `CONTINUE` ou `END DO`. * Variables d'indice et instructions de contrôle réels simple ou double précision.
- Instruction `ASSIGN` et l'instruction `GOTO` assigné.
- `GO TO` calculé remplacé par une construction `CASE`,
- `ASSIGN` d'une étiquette de `FORMAT`.

F90/95 : Caractéristiques obsolètes

- l'instruction RETURN secondaire remplacé par une construction CASE,
- Instruction PAUSE.
- l'instruction DATA placée au sein des instructions exécutables,
- Descripteur d'édition H.
- le type `character(len=*)` de longueur implicite en retour de fonction,
- le type `character*` dans les déclarations.
- fonction instruction remplacé par les fonctions internes,

F90/95 :Évolution vers la norme Fortran95

Le processus de normalisation se poursuit mais les apports de la norme Fortran95 sont mineurs. Des extensions sont proposés pour une adaptation aux machines parallèles distribuées.

- Instruction et construction FORALL,
- les attributs PURE et ELEMENTAL pour les procédures sans effet de bord,
- la fonction intrinsèque NULL() pour forcer un pointeur à l'état non associé,
- fonction utilisateur dans les expressions de spécification,
- extension de la construction WHERE : blocs imbriqués,
- fonction intrinsèque `CPU_TIME`,
- libération des tableaux dynamiques locaux n'ayant pas l'attribut SAVE,
- initialisation par défaut pour les objets de type dérivé,
- extension des fonctions intrinsèques CEILING, FLOOR,MAXLOC, MINLOC,
- Commentaire dans les spécifications de NAMELIST,
- modifications pour s'adapter à IEEE 754/854.

Quelques règles de codage

Il est recommander d'adopter une charte pour uniformiser les développements

- Mots clés fortran, fonctions intrinsèques, types utilisateurs, doivent être en majuscule.
- Le nom des variables première lettre en majuscule puis les autres en minuscule
- Pensez à indenter le corps des unités de programmes, des blocs de contrôles, les blocs interface, ...
- Commenter vos codes : description en entête des procedures, et dans le code à l'aide de !

Fortran 2003

- Intégration dans le système `GET_ARG`
- Interopérabilité avec le C.
- polymorphisme (des variables, des pointeurs et des procédures) `CLASS`, `SELECT TYPE`,
- l'Héritage (type dérivés, procédures)
- Surcharge de procédures

Bibliographie

- C. Delannoy, *Programmer en Fortran 90*, Eyrolles, 1998.
- P. Lignelet, *Manuel complet du langage Fortran 90 et Fortran 95, Calcul intensif et génie logiciel*, Masson, 1996.
- M. Metcalf and J. Reid, *fortran 90/95 explained*, Oxford sciences publications, 1996.
- P. Corde et H. Delouis *Cours Fortran 90/95 de l'Idris (2ème niveau)* http://webserv2.idris.fr/data/cours/lang/fortran/choix_doc.html
- M. Dubesset et J. Vignes, *les spécificités du fortran 90*, Editions technip, 1993.

Précision

- Fichier *precision1.f90* : Ce programme permet de tester des opérations sur et entre différents types ainsi que la conversion d'un réel en un entier. Il permet également d'observer les différences entre `kind=4` et `kind=8` ansi que les affectations dans ces deux types :
  - `x=0.9` (simple précision)
  - `x=9.0d-1` (double précision)

  Des perturbations numériques apparaissent dès qu'un réel simple précision est affectée dans une variable double précision. En résumé, les sous-types définis par `kind` ne doivent pas être négligés.
- Fichier *precision2.f90* : Programme de test de test de la précision machine présenté dans le cours

- Fichier *where.f90* : Ce programme présente différentes techniques d'initialisations et d'accès à un tableau. En particulier, il présente une utilisation des vecteurs anonymes et des pseudo-boucles. Une utilisation de la fonction intrinsèque `where` est proposée. L'exemple que nous considérons ici est de calculer la racine carrée des éléments d'un tableau s'ils sont positifs avec un cutt-off à zéro pour les éléments négatifs.

- Fichiers *main.f90 lect.f90 integ.f90 fonc.f90* : Ce programme est un exemple de procédure passée en argument d'une fonction. Le programme lit une liste de point, `X`, contenu dans le fichier LISTE. Cette procédure de lecture appartient au module `lect`. On propose ensuite d'integrer une fonction sur l'intervalle `[MINVAL(X),MAXVAL(X)]`. Cette fonction à intégrer est donnée dans le module `fonc` et s'appelle `func` avec un argument `m` permettant de choisir une fonction parmis plusieurs. `fonc` est passé en argument de la fonction `integ`. Les autres arguments de `integ` sont `X` (pour les bornes de l'intervalle d'intégration), `m` (pour le choix de la fonction à intégrer) et `meth` (pour le choix de la méthode d'intégration). Il est important de noter la présence de l'interface (obligatoire) de description de la fonction passée en argument dans la fonction `integ`.

- Fichiers *frac_mod.f90 surcharge.f90* : Ce programme présente un exemple de création de deux nouveaux opérateurs `.plus.` et `.mult.` définis dans le module `op_objet`. Ces opérateur permettent l'addition et la multiplication entre deux variables dont le type est défini par deux champs. Ce type permet de représenter les fractions.
- Fichiers *frac_mod.f90 comp_mod.f90 surcharge.f90* : Ce programme est identique au précédent mais cette fois, le type est abstrait. Les champs qui le composent ne sont accessibles que dans le module `op_objet`. Nous avons donc ajouté dans le module des procédures permettant d'initialiser de nouvelles variables et de les afficher à l'écran. Le fichier *comp_mod.f90* est identique à *frac_mod.f90* mais pour des nombres complexes. Cela permet avec le même programme principal de traiter des fractions ou des nombres complexes suivants les besoins de l'utilisateur.
- Fichiers *frac_mod.f90 comp_mod.f90 surcharge.f90* : Ce programme est identique au précédent mais avec une surcharge des opérateurs + et *.

Résolution Numérique d'une LdC 1D.

- La forme continue du problème: $\boldsymbol{u}(x,t)$

$$\partial_t \boldsymbol{u} + \partial_x \left(f(\boldsymbol{u}) \right) = 0, \quad \boldsymbol{u}(x,0) = \boldsymbol{u}_0(x), \quad \boldsymbol{u}(x+L,t) = \boldsymbol{u}(x,t)$$

- Maillage structuré: $t^n = n\delta t$, $x_j = j\delta x$, $n = 0, Nmax$ et $j = 1, Ns$

$$x_1 = \delta x, \quad x_{Ns} = L, \quad \delta x = \frac{L}{Ns}$$

- La solution approchée est définie par:

$$\boldsymbol{v}(x,t) = \boldsymbol{v}_i^n \quad \forall (x,t) \in]x_{i-\frac{1}{2}},\ x_{i+\frac{1}{2}}[\times [t^n,\ t^{n+1}[,$$

$$\boldsymbol{v}_i^0 = \frac{1}{\delta x} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} \boldsymbol{u}_0(x) dx$$

Schéma et Flux Numériques.

- Schéma conservatif:

$$\boldsymbol{v}_i^{n+1} = \boldsymbol{v}_i^n - \frac{\delta t}{\delta x}\left(\boldsymbol{\phi}_{i+\frac{1}{2}} - \boldsymbol{\phi}_{i-\frac{1}{2}}\right)$$

- Flux de Lax-Friedrichs :

$$\boldsymbol{\phi}_{i+\frac{1}{2}} = \boldsymbol{\phi}^{LF}(\boldsymbol{v}_i^n, \boldsymbol{v}_{i+1}^n) = \frac{1}{2}\left(\mathbf{f}_{i+1}^n + \mathbf{f}_i^n + \frac{\delta x}{\delta t}\left(\boldsymbol{v}_i^n - \boldsymbol{v}_{i+1}^n\right)\right)$$

- Flux de Murman-Roe :

$$\boldsymbol{\phi}_{i+\frac{1}{2}} = \boldsymbol{\phi}^{MR}(\boldsymbol{v}_i, \boldsymbol{v}_{i+1}) = \frac{1}{2}\left(\mathbf{f}_{i+1}^n + \mathbf{f}_i^n + \left|\frac{\Delta \mathbf{f}_{i+\frac{1}{2}}^n}{\Delta \boldsymbol{v}_{i+\frac{1}{2}}^n}\right|\left(\boldsymbol{v}_{i+1}^n - \boldsymbol{v}_i^n\right)\right)$$

avec $\Delta X_{i+\frac{1}{2}}^n = X_{i+1}^n - X_i^n$

Schéma et Flux Numériques.

- Flux de Lax-Wendroff :

$$\boldsymbol{\phi}^{LW}(\boldsymbol{v}_i, \boldsymbol{v}_{i+1}) = \frac{1}{2}\left(\mathbf{f}^n_{i+1} + \mathbf{f}^n_i - \frac{\delta x \Delta \mathbf{f}^n_{i+\frac{1}{2}}}{\delta t \Delta \boldsymbol{v}^n_{i+\frac{1}{2}}}\left(\mathbf{f}^n_{i+1} - \mathbf{f}^n_i\right)\right)$$

- Condition de stabilité approchée:

$$\max\left|\frac{\delta t}{\delta x}\frac{\Delta \mathbf{f}^0_{i+\frac{1}{2}}}{\Delta \boldsymbol{v}^0_{i+\frac{1}{2}}}\right| = CFL < \frac{1}{2} \Longrightarrow \delta t = \frac{\delta x CFL}{\max\left|\frac{\Delta \mathbf{f}^0_{i+\frac{1}{2}}}{\Delta \boldsymbol{v}^0_{i+\frac{1}{2}}}\right|}$$

Algorithme.

- $\mathbf{f}(\boldsymbol{v})$ et $\boldsymbol{u}_0(x)$ des fonctions données.
- L, Ns, CFL, T paramètres donnés, $Tl = 0$.
- Initialisations : $\delta x = \frac{L}{Ns}$, $x_j = j\delta x$ pour $j = 1, Ns$.
- Initialisations : $\boldsymbol{v}_i^0 \simeq \boldsymbol{u}_0(i\delta x)$ et $\mathbf{f}_i^0 \simeq \mathbf{f}(\boldsymbol{v}_i^0)$ pour $i = 1, Ns$.
- Initialisations : $\delta t = \dfrac{\delta x CFL}{\max \left| \dfrac{\Delta \mathbf{f}^0_{i+\frac{1}{2}}}{\Delta \boldsymbol{v}^0_{i+\frac{1}{2}}} \right|}$,

  $Nmax \simeq INTEGER(T/\delta t)$.
- Boucle en temps

$$\boldsymbol{v}_i^{n+1} = \boldsymbol{v}_i^n - \frac{\delta t}{\delta x}\left(\boldsymbol{\phi}_{i+\frac{1}{2}} - \boldsymbol{\phi}_{i-\frac{1}{2}}\right)$$

Boucle en temps.

- Boucle en temps: Pour $n = 0, Nmax + 1$
  - Calculer les flux $\boldsymbol{\phi}_{i+\frac{1}{2}}$ pour $i = 1, Ns - 1$.
  - Conditions aux limites périodiques: $\boldsymbol{\phi}_{\frac{1}{2}} = \boldsymbol{\phi}_{Ns+\frac{1}{2}} = \boldsymbol{\phi}(\boldsymbol{v}^n_{Ns}, \boldsymbol{v}^n_1)$.
  - Evolution en temps: $\boldsymbol{v}^{n+1}_i = \boldsymbol{v}^n_i - \frac{\delta t}{\delta x}\left(\boldsymbol{\phi}_{i+\frac{1}{2}} - \boldsymbol{\phi}_{i-\frac{1}{2}}\right)$.
  - $Tl = Tl + \delta t$, si $T - Tl < \delta t$ alors $\delta t = T - Tl$

Mise en œuvre En fortran 90.

Paramètre	Caractéristiques en F90			
	Type	Rang	Dim	Variable
i	INTEGER	1	1	i
j	INTEGER	1	1	j
Ns	INTEGER	1	1	Ns
n	INTEGER	1	1	n
$nmax$	INTEGER	1	1	Nmax
L	REAL	1	1	L
T	REAL	1	1	T
Tl	REAL	1	1	Tl
CFL	REAL	1	1	CFL

Mise en œuvre En fortran 90.

Paramètre	Caractéristiques en F90			
	Type	Rang	Dim	Nom
δx	REAL	1	1	Dx
δt	REAL	1	1	Dt
x_j	REAL	1	Ns	Xp
$\phi_{j+\frac{1}{2}}$	REAL	1	0:Ns	FluxN
$\boldsymbol{u}_j^n$	REAL	1	1:Ns	Rho
$\boldsymbol{u}_j^{n+1}$	REAL	1	1:Ns	RhoNew

Mise en œuvre En fortran 90.

Paramètre	Caractéristiques en F90			
	Type	Rang	Dim	Nom
$\max \left\| \dfrac{\Delta \mathbf{f}^0_{i+\frac{1}{2}}}{\Delta \boldsymbol{v}^0_{i+\frac{1}{2}}} \right\|$	REAL	1	1	Cmax
$\boldsymbol{u}_0(x)$	REAL FUNCTION	1	1	RhoInit
$\mathbf{f}$	REAL FUNCTION	1	1	Flux
$\boldsymbol{\phi}^{LF}(\boldsymbol{v}_i, \boldsymbol{v}_j)$	REAL FUNCTION	1	1	FluxLF
$\boldsymbol{\phi}^{LW}(\boldsymbol{v}_i, \boldsymbol{v}_j)$	REAL FUNCTION	1	1	FluxLW
$\boldsymbol{\phi}^{MR}(\boldsymbol{v}_i, \boldsymbol{v}_j)$	REAL FUNCTION	1	1	FluxMR

Structuration du logiciel en fortran 90.

Organisation:

- Un module `LesFonctions` de fonctions (réutilisable dans d'autres contextes) contenant les fonctions .
- Un programme principal `LdcNonLineaire` avec un mot clé (`RootName`) donné à l'exécution du programme.
- Lectures des données dans un fichier formaté : (`RootName`)`.data`.
- Sauvegardes dans des fichiers de la forme (`RootName`)_(n)`.xmgr`.
  où n est le numéro de l'itération à laquelle la sauvegarde a lieu.

Fonctions Intrinèques Utililes.

- `TRIM` : Construit une nouvelle chaîne de caractère en Supprimant les blancs à la fin d'une autre chaîne de caractère. `NewName = TRIM(OldName)`
- `AJUSTL` : Supprime les blancs au début d'une chaîne de caractère. `NewName = AJUSTL(OldName)`
- `LEN_TRIM` : Donne la taille d'une chaîne de caractère, ne tenant pas en compte des blancs à la fin. `il=LEN_TRIM(OldName)`
- `WRITE` : utilisé ici pour transformer un entier en chaîne de caractère. `WRITE(OldName,*) n`

Procédure de construction de Nom de fichier.

La première procédure `TheNewName` que nous décrivons est celle qui construit un nom de fichier `NewName` à partir d'une chaîne de caractère `RootName` et d'un nombre entier n représenté sur `Nm` caractères et complété au besoin par des zéros au début.

Procédure TheNewName.

```
SUBROUTINE TheNewName(Name, n, Nm, NewName)
 CHARACTER(LEN=*), INTENT(IN)  :: Name
 INTEGER         , INTENT(IN)  :: n, Nm
 CHARACTER(LEN=*), INTENT(OUT) :: NewName
 INTEGER                 :: l , m
 CHARACTER(LEN=20)       :: NB, Ze

 Ze ="00000000000000";  WRITE(NB, *) n
 NB = ADJUSTL(NB);  l= LEN_TRIM(NB) ; m=Nm-l
 IF( l > 0 ) THEN
  NewName=TRIM(Name)//"_"//Ze(1:m)//TRIM(NB)
 ELSE
  NewName=TRIM(Name)//"_"//TRIM(NB)
 END IF
END SUBROUTINE TheNewName
p
```

Fonction Flux de Roe.

```
REAL FUCTION  RoeFlux(Vi, Vj, Flux)
 REAL    , INTENT(IN)  :: Vi, Vj
 REAL    , EXTERNAL    :: Flux

END SUBROUTINE TheNewName
```

Déclarations En fortran 90.

```
INTEGER :: Ns, Nmax,  n, i,  j
REAL    ::  L,   Dx, Dt, T, Tl, Cmax, CFL

REAL, DIMENSION(:), ALLOCATABLE :: Rho, RhoNew
REAL, DIMENSION(:), ALLOCATABLE ::  Xp, FluxN
```

Mise en œuvre En fortran 90.

Sauvegarde de la solution au temps final:

```
Open(10,FILE=Final.xmgr)
DO i = 1, Ns
 WRITE(10, ‘(1x,2(EN15.8,1x) )’) X(i), V(i)
END DO
CLOSE(10)
```

Ensemble des fichiers utilisés pour les travaux pratiques.

Travaux Pratiques

Familiarisation aux subtilités du Fortran 90/95.

Utilisation des différents types d'Interfaces

Travaux Pratiques

Exemples de mise en œuvre de la modularité.

Performances en fonction de la stratégie de programmation

Travaux Pratiques

Exemples divers.