

Cours de programmation: Fortran 90

Alexandre Mayer

Laboratoire de Physique du Solide
Université de Namur
Rue de Bruxelles 61, 5000 Namur, Belgique
alexandre.mayer@unamur.be

<http://perso.unamur.be/~amayer>

Table des matières

- ◆ Notions de base
- ◆ Les structures de contrôle
- ◆ Les opérations de lecture et d'écriture
- ◆ Les tableaux
- ◆ Les procédures
- ◆ Les procédures: notions avancées
- ◆ Les modules
- ◆ Les modules: notions avancées
- ◆ Les pointeurs
- ◆ La librairie DFLIB
- ◆ Notions élémentaires d'optimisation
- ◆ Exécution en lignes de commandes



Notions de base

Chapitre 1



Workspace 'Exemple1': 1 proje
Exemple1 files
Exemple1.f90

```
Program Main
  implicit none
  real :: x, y, surface

  print *, "Calcul de la surface d'un rectangle"
  print *

  print '(a,$)', " Entrez x : " ; read *, x
  print '(a,$)', " Entrez y : " ; read *, y

  surface = x * y      ! calcul de la surface

  print *
  print *, "La surface du rectangle &
    &vaut : ", surface

end Program Main
```

"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre1\Debug\Exemple..."

```
Calcul de la surface d'un rectangle
Entrez x : 5
Entrez y : 6

La surface du rectangle vaut :    30.00000
Press any key to continue_
```

-----Configuration: Exemple1 - Win32 Debug-----
Exemple1.exe - 0 error(s), 0 warning(s)

Build Debug Find in Files 1 Find in Files 2

Le format libre

- ◆ 132 caractères par ligne
- ◆ Le ! signifie que le reste de la ligne est du commentaire.
- ◆ Une ligne terminée par & (ampersand) se poursuit sur la ligne suivante. Le & peut être reproduit sur la ligne suivante.
- ◆ Le ; sépare deux instructions sur la même ligne.
- ◆ Des espaces peuvent être introduits entre les éléments de langage.

Warning: l'extension du fichier doit être .f90 pour travailler en format libre. Un fichier en .f est traité en format fixe.

Jeu de caractères

- ◆ Alphanumériques: A-Z, a-z, 0-9, _
- ◆ Autres: = + * / () , . ' " : ; ! & % < > \$?
et l'espacement (space)

Le Fortran ne distingue pas les majuscules des minuscules sauf dans les chaînes de caractères.

Les noms des variables peuvent être composés d'une suite de 1 à 31 caractères alphanumériques dont le premier est une lettre.

Structure d'un programme

```
Program < Nom du Programme >  
  
:  
: ! Instructions non exécutables  
:  
  
:  
: ! Instructions exécutables  
:  
  
end Program < Nom du Programme >
```

- Instructions non exécutables: déclaration des objects que l'on va utiliser dans le programme. Ces infos ne sont utilisées que lors de la compilation.
- Instructions exécutables: opérations mathématiques, lecture/écriture de fichiers, appel à des fonctions ou sous-routines, etc.
Les commentaires et lignes blanches ne sont pas traités par le compilateur.

Les types de données

CHARACTER des chaînes d'un ou plusieurs caractères

LOGICAL des valeurs booléennes ou logiques, qui ne peuvent prendre que deux valeurs, vrai ou faux (*true* ou *false*)

INTEGER des nombres entiers, qui peuvent prendre toutes les valeurs positives ou négatives entre des limites qui dépendent de la machine

REAL des nombres réels, qui peuvent prendre des valeurs positives ou négatives, avoir une partie fractionnaire, et prendre des valeurs absolues très grandes ou très petites

COMPLEX des nombres complexes composés d'une partie réelle et d'une partie imaginaire, toutes deux de type réel

Les limites de la représentation

Entiers:

$$-2.147.483.648 \leq i \leq 2.147.483.647$$

(codage en 4 octets)

Réels simple précision:

$$1.2 \cdot 10^{-38} \leq |x| \leq 3.4 \cdot 10^{38}$$

(7 chiffres significatifs, codage en 4 octets)

Réels double précision:

$$2.2 \cdot 10^{-308} \leq |x| \leq 1.8 \cdot 10^{308}$$

(16 chiffres significatifs, codage en 8 octets)

Les constantes littérales

Les constantes littérales ont des valeurs fixes.

Exemples

2.0	!	REAL
-3.2E06	!	REAL
1.2E-06	!	REAL
176	!	INTEGER
.TRUE.	!	LOGICAL
'F90'	!	CHARACTER

Les types implicites

Si dans un code apparaissent des noms qui n'ont pas été définis par des instructions de spécification, le type de la variable dépend de sa première lettre

- I, J, K, L, M, N représentent les entiers
- les autres lettres définissent les réels

L'utilisation des types implicites peut donner lieu à des erreurs non détectées par le compilateur et qui sont difficiles à trouver.

Il vaut mieux renoncer à la règle de définition implicite et définir explicitement toutes les variables en plaçant l'instruction de spécification

IMPLICIT NONE

avant toutes les autres.

La déclaration des variables numériques et logiques

$\langle type \rangle$ [$, \langle attribut(s) \rangle$] [$::$] $\langle variable(s) \rangle$ [$= \langle value \rangle$]

Si dans l'instruction de spécification apparaissent $\langle attribut(s) \rangle$ ou $= \langle value \rangle$, le $::$ est obligatoire.

Les *attributs* possibles sont:

- PARAMETER, SAVE, INTENT, POINTER, TARGET, ALLOCATABLE, DIMENSION, PUBLIC, PRIVATE, EXTERNAL, INTRINSIC, OPTIONAL.

La déclaration des variables numériques et logiques

Exemples

```
REAL a
```

```
REAL :: c,d,e           ! Specification de plusieurs  
                        ! variables par une instruction
```

```
INTEGER :: a_int=10      ! Initialisation
```

```
INTEGER, PARAMETER :: maxat=100 ! Constante
```

```
INTEGER, PARAMETER :: at_xyz=3*maxat
```

```
REAL, DIMENSION(10) :: a_vec,b_vec
```

```
LOGICAL :: converged=.FALSE.
```

```
REAL :: ax, bx=1.0
```

```
COMPLEX :: z=(1.0,1.0)
```

La déclaration des constantes

Exemple

```
INTEGER, PARAMETER :: maxat=100 ! Constante
```

La spécification d'une valeur constante est conseillée

- si la variable ne change pas lors de l'exécution du programme
- pour rendre un code plus lisible (par exemple, `PI`, `DEUX_PI`, ...)
- pour rendre un code plus facile à modifier, si la valeur d'une *constante* (les dimensions d'une matrice, par exemple) risque d'être changée; le modification d'une ligne suffit pour changer tout un programme

La déclaration des chaînes de caractères

`CHARACTER[(LEN=< longueur >)] [, < attribut(s) >] [::] < variable(s) > [= < value >]`

Exemples

```
CHARACTER(LEN=10) :: nom
```

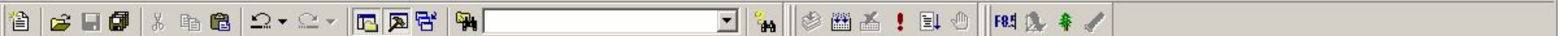
```
CHARACTER :: oui_ou_non
```

```
CHARACTER(LEN=5) :: ville='NAMUR'
```

```
CHARACTER(LEN=8) :: town='Namur'           ! Padded right
```

```
CHARACTER(LEN=8) :: capitale="BRUXELLES" ! Truncated
```

```
CHARACTER(LEN=*), PARAMETER :: lang='F90' ! Assumed length
```



Workspace 'Exemple2': 1 proje
Exemple2 files
Exemple2.f90

```
Program Main
character(len=8) :: town = "Namur"
character(len=8) :: capital = "Bruxelles"
character(len=*) , parameter :: lang = "F90"

print *, "-->//town//<--"
print *, "-->//capital//<--"
print *, "-->//lang//<--"

end Program Main
```

"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre1\Debug\Exemple..."

```
-->Namur <--
-->Bruxelle<--
-->F90<--
Press any key to continue
```

-----Configuration: Exemple2 - Win32 Debug-----
Exemple2.exe - 0 error(s), 0 warning(s)

Build Debug Find in Files 1 Find in Files 2

Les opérateurs numériques intrinsèques

- * / opérateurs de multiplication et division; opérateurs *dyadiques*.
2 * 3, 9.0/2.0
- + - opérateurs d'addition et soustraction; opérateurs monadiques ou dyadiques.
-5, 7-4+2
- ** opérateur d'exponentiation, opérateur dyadique.
3**2

Les opérateurs numériques intrinsèques

Si plusieurs opérateurs arithmétiques sont présents dans une expression, l'ordinateur effectue, dans l'ordre

- les exponentiations
- les multiplications et les divisions, en allant de gauche à droite
- les additions et les soustractions, en allant de gauche à droite

En cas de doute, *utiliser les parenthèses*.

Attention:

- Le résultat de la division entre deux entiers est un entier, la partie entière du quotient exact. L'expression $5/2$ a pour valeur 2.
- On ne peut pas diviser par zéro.

Les opérateurs de comparaison

Les opérateurs de comparaison ont pour opérandes des expressions numériques et pour résultat une valeur logique.

== ou **.eq.** est égal à

/= ou **.ne.** n'est pas égal à

> ou **.gt.** plus grand que

>= ou **.ge.** plus grand ou égal

< ou **.lt.** plus petit que

<= ou **.le.** plus petit ou égal

Seuls les opérateurs **==** et **/=** s'appliquent entre deux opérandes complexes.

Les opérateurs de comparaison s'évaluent après les opérateurs arithmétiques et avant les opérateurs logiques.

Les opérateurs logiques intrinsèques

Une expression *logique* ne peut donner comme résultat que `.TRUE.` ou `.FALSE.`

p	q	.not.p	p.and.q	p.or.q	p.eqv.q	p.neqv.q
t	t	f	t	t	t	f
t	f	f	f	t	f	t
f	t	t	f	t	f	t
f	f	t	f	f	t	f

Les opérateurs caractères intrinsèques



Pour les chaînes de caractères existent les opérateurs:

// concaténation (mise bout-à-bout de deux chaînes)

== et /= comparaison

Le Fortran distingue les minuscules des majuscules dans les chaînes de caractères.

Morceaux de chaînes de caractères

```
CHARACTER(LEN=*) , PARAMETER :: chaine="pqrstuv"
```

`chaine(2:2)` signifie le deuxième caractère, 'q'

`chaine(3:6)` signifie la chaîne allant du troisième au sixième caractère, 'rstu'

`chaine(:3)` signifie les trois premiers caractères de `chaine`

`chaine(6:)` signifie les derniers caractères de `chaine` à partir du sixième, 'uv'

Autres opérations sur les chaînes de caractères

Program Main

```
character(len=12) :: mot="physique"
```

```
print *, len(mot)           ! longueur de la variable (!)
```

```
print *, trim(mot)         ! supprime les blancs à la fin
```

```
print *, trim(adjustl(mot)) ! pousse à gauche et  
! supprime les blancs à la fin
```

```
print *, mot(1:3)          ! 3 premières lettres
```

```
print *, mot(1:3)//mot(4:) ! concaténation
```

```
print *, index(mot,"que")  ! donne 6
```

```
end Program Main
```



Workspace 'Exemple3': 1 proje

- Exemple3 files
 - Exemple3.f90

```

Program Main
character(len=12) :: mot="physique", num

print *, "-->"// mot // "<--"
print *, "-->"// trim(mot) // "<--"
print *, "-->"// adjustl(mot) // "<--"
print *, "-->"// trim(adjustl(mot)) // "<--"
print *, "-->"// adjustr(mot) // "<--"
print *, "-->"// mot(1:3) // "<--"
print *, "-->"// mot(1:3)//mot(4:) // "<--"
print *, len(mot)
print *, len(trim(adjustl(mot)))
print *, index(mot,"que")
print *
write(num,*) 3
mot = "Exemple"//trim(adjustl(num))//".dat"
print *, mot

end Program Main

```

```

-->physique <--
-->physique<--
-->physique <--
-->physique<--
--> physique<--
-->phy<--
-->physique <--
      12
      8
      6

Exemple3.dat
Press any key to continue

```

```

-----Configuration: Exemple3 - Win32 Debug-----
Exemple3.exe - 0 error(s), 0 warning(s)

```


Les opérations d'affectation

L'affectation est l'instruction exécutable qui permet d'affecter une valeur à une variable ou de changer sa valeur, si elle en avait déjà une.

```
i = 4*j + k + 6
```

```
a = 1.0/b
```

```
m = m + 1
```

- A gauche du signe =, on écrit une variable
- A droite du signe =, on écrit une expression (contenant des constantes ou des variables auxquelles une valeur a déjà été donnée).

Une variable définie sans valeur initiale a une valeur indéterminée en début d'exécution.

Les types de données paramétrisés

Les types de données paramétrisés permettent d'assurer la portabilité de la précision numérique des programmes écrits en Fortran 90. Les compilateurs supportent au moins deux sortes de réels, identifiés par un nombre entier (en général le nombre d'octets utilisés pour la représentation de ces réels).

```
REAL(kind=4) :: x ! simple précision
```

```
REAL(kind=8) :: y ! double précision
```

```
INTEGER(kind=4) :: i
```

Le *kind* de constantes littérales réelles se note en ajoutant `_` et le numéro du kind derrière la constante. Par exemple, `3.4_8`.

Il faut noter que tous les compilateurs ne codent pas forcément la simple et la double précision avec des kinds de 4 et de 8 (mais je n'ai jamais rencontré de tels compilateurs).

Les types de données paramétrisés

Pour faciliter la conversion de tout un programme dans un autre *kind* de réels, on a intérêt à définir une constante pour le *kind*.

```
Program Main
implicit none
integer, parameter :: r=4
real(kind=r) :: x, y=2._r
...
end Program Main
```

Les types de données paramétrisés

La meilleure façon de définir un *kind* est d'utiliser les fonctions intrinsèques **SELECTED_REAL_KIND** et **SELECTED_INT_KIND**.

- **SELECTED_INT_KIND**(r): la valeur de la première sorte d'entiers disponible capable de représenter les entiers de -10^r à 10^r ($r \equiv range$)
- **SELECTED_REAL_KIND**(p, r): la valeur de la première sorte de réels disponible capable de représenter les réels ayant au moins p chiffres décimaux significatifs de -10^r à 10^r ($p \equiv precision$, $r \equiv range$)

Si la précision demandée n'est pas disponible, **SELECTED_REAL_KIND** et **SELECTED_INT_KIND** retourne la valeur -1, d'où messages d'erreur à la compilation des instructions qui utilisent les paramètres.

Les types de données paramétrisés

Dans l'exemple suivant, x et y ont 15 chiffres décimaux significatifs. Ils peuvent prendre des valeurs comprises entre -10^{70} et 10^{70} . Ces spécifications requièrent la double précision.

```
Program Main
implicit none
integer, parameter :: r=selected_real_kind(15,70)
real(kind=r) :: x, y=2._r

...

end Program Main
```

Opérations entre différents kinds

En cas d'opération entre objets de type ou de kind différents, l'ordinateur convertit les deux opérandes vers des valeurs de type et de kind identiques. Suivant la situation rencontrée, les conversions suivantes sont ainsi réalisées:

- les entiers sont convertis en réels ou en complexes
- les réels sont convertis en complexes
- les réels ou les complexes sont convertis dans le kind le plus élevé.

Dans une affectation, l'expression (à droite) est d'abord évaluée, puis convertie dans le type et le kind de la variable (à gauche).

Les nombres complexes

Une donnée numérique complexe consiste en deux réels: sa partie réelle et sa partie imaginaire. Les instructions de spécification, les opérateurs, les expressions et les affectations sont analogues à ceux des réels.

`integer,parameter :: r=8`

`complex(kind=r) :: z`

`complex(kind=r),parameter :: i=(0._r,1._r)`

`z=(1._r,0.5_r)*i`

Les nombres complexes

Program Main

```
complex(kind=4) :: z  
real(kind=4) :: x=0., y=1.
```

```
z=cmplx(x,y)      ! assignation  
z=(0.,1.)
```

```
print *, " Partie réelle de z ", real(z)  
print *, " Partie imaginaire de z ", aimag(z)  
print *, " Complexe conjugué de z ", conjg(z)
```

```
end Program Main
```


Les structures « composites »

```
program Main
```

```
type Atom
```

```
    character(len=2) :: symbol
```

```
    integer :: Z
```

```
    real :: A
```

```
end type
```

```
type(Atom) :: Carbon
```

```
Carbon=Atom("C",6,12.0107)    ! affectation « globale »
```

```
Carbon%symbol = "C"          ! affectation par composantes
```

```
Carbon%Z = 6
```

```
Carbon%A = 12.0107
```

```
print *, Carbon%Z
```

```
end program Main
```

Les structures « composites »

```
program Main
```

```
type Atom
```

```
    character(len=2) :: symbol
```

```
    integer :: Z
```

```
    real :: A
```

```
end type
```

```
type(Atom),dimension(1:118) :: PeriodicTable
```

```
PeriodicTable(6)=Atom("C",6,12.0107) ! affectation « globale »
```

```
PeriodicTable(6)%symbol = "C" ! affectation par composantes
```

```
PeriodicTable(6)%Z = 6
```

```
PeriodicTable(6)%A = 12.0107
```

```
print *, PeriodicTable(6)%Z
```

```
end program Main
```

Les structures de contrôle

Chapitre 2

L'instruction IF

L'instruction IF permet de soumettre l'exécution d'une ou plusieurs instructions à une condition.

si <condition> → <instructions exécutables>

S'il n'y a qu'une seule instruction à exécuter, on peut utiliser la forme compacte du IF comme dans l'exemple suivant:

```
IF (x>=0.) print *, « x est positif »
```

S'il y a plusieurs instructions à exécuter, on doit utiliser un bloc IF:

```
IF (x>=0.) THEN  
    print *, « x est positif »  
    print *, « x = », x  
END IF
```

L'instruction IF

On peut utiliser le ELSE dans un bloc IF pour proposer une alternative:

```
IF (x >= 0.) THEN
    print *, « x est positif »
ELSE
    print *, « x est strictement négatif »
END IF
```

```
IF (x > 0.) THEN
    print *, « x est strictement positif »
ELSE IF (x < 0.) THEN
    print *, « x est strictement négatif »
ELSE
    print *, « x est nul »
END IF
```

L'instruction IF

Seul le bloc correspondant à la première condition vraie est exécuté. Si aucune condition n'est vraie, le bloc d'instructions qui suit le ELSE est exécuté.

On peut utiliser autant d'instructions ELSE IF que l'on veut. Par ailleurs, les blocs IF peuvent être imbriqués. Il est également possible d'associer un nom à un bloc IF:

```
test: IF (x >= 0.) THEN
    print *, « x est positif »
ELSE test
    print *, « x est strictement négatif »
END IF test
```

Finalement, les formes compactes ELSEIF et ENDIF peuvent être utilisées à la place de ELSE IF et END IF.

Les boucles DO

L'instruction DO permet de répéter l'exécution d'un bloc d'instructions.

$n \times \{ \text{instructions exécutables} \}$

Boucle DO infinie:

DO

print *, « Je ne peux pas parler en classe. »

END DO

Il y a deux façons « propres » de sortir d'une telle boucle:

STOP : cette instruction arrête l'exécution du programme

EXIT: cette instruction permet de sortir du bloc DO dans lequel elle se trouve. Le programme continue alors à la ligne qui suit le END DO.

L'instruction CYCLE permet de revenir au début de la boucle, c'est-à-dire à la ligne qui suit le DO du bloc en question.

Les boucles DO

Il est possible de donner un nom à une boucle DO:

```
boucle: DO
    print *, « Je ne peux pas parler en classe. »
END DO boucle
```

La structure complète d'une boucle DO est donc:

```
boucle: DO
    ...
    if (<condition>) CYCLE
    ...
    if (<condition>) EXIT
END DO boucle
```


Les boucles DO

On peut imbriquer des boucles DO. Les instructions EXIT et CYCLE se rapportent alors à la boucle la plus intérieure dans laquelle elles se trouvent, sauf si elles sont suivies du nom d'une boucle.

```
boucle_ext: DO
```

```
    ! On arrive ici avec CYCLE boucle_ext
```

```
    boucle_int: DO
```

```
        ! On arrive ici avec CYCLE
```

```
        ...
```

```
        if (<condition>) EXIT
```

```
        if (<condition>) EXIT boucle_ext
```

```
        if (<condition>) CYCLE
```

```
        if (<condition>) CYCLE boucle_ext
```

```
    END DO boucle_int
```

```
    ! On arrive ici avec EXIT
```

```
END DO boucle_ext
```

```
! On arrive ici avec EXIT boucle_ext
```

Le DO WHILE

Une autre façon de limiter le nombre d'itérations est d'utiliser un DO WHILE:

```
DO WHILE (<condition>)
```

```
...
```

```
END DO
```

La boucle s'arrête dès que la condition (testée au début de chaque itération) s'avère fausse. Il est par ailleurs toujours permis d'utiliser les instructions EXIT et CYCLE.

La boucle DO à nombre fixé d'itérations

Si l'on connaît le nombre d'itérations à réaliser, on utilisera plutôt une boucle du type:

```
DO <integer> = <début>, <fin>, <incrément>  
...  
END DO
```

La variable entière <integer> peut être utilisée à l'intérieur de la boucle, mais elle ne peut y être modifiée. Cette variable vaut <début> à la première itération. Elle est augmentée de <incrément> lors de chaque nouvelle itération. Le programme sort de la boucle lorsque la valeur <fin> est dépassée. Si <incrément> n'est pas spécifié, il vaut 1 par défaut. Si <fin> < <début>, il faut spécifier un <incrément> négatif (sinon le programme n'entre pas dans la boucle).

La boucle DO à nombre fixé d'itérations

integer :: i

```
DO i=1, 5  
    print *, « i=», i  
END DO
```

i= 1
i= 2
i= 3
i= 4
i= 5

```
DO i=1, 5, 2  
    print *, « i=», i  
END DO
```

i= 1
i= 3
i= 5

```
DO i=5, 1, -2  
    print *, « i=», i  
END DO
```

i= 5
i= 3
i= 1

```
DO i=5, 1  
    print *, « i=», i  
END DO
```

La structure SELECT CASE

La structure SELECT CASE permet de choisir entre différents blocs d'instructions à réaliser, en fonction de la valeur d'une expression.

```
nom: SELECT CASE (<expression>)  
CASE (<choix 1>) nom  
    ...  
CASE (<choix 2>) nom  
    ...  
CASE DEFAULT nom  
    ...  
END SELECT nom
```

<expression> peut être une expression de type caractère, entier, ou logique. On peut utiliser des SELECT CASE imbriqués. Il n'est pas obligatoire de donner un nom à une structure SELECT CASE.

La structure SELECT CASE

integer :: i

SELECT CASE (i)

CASE (0)

... ! s'exécute si $i=0$

CASE (:-1)

... ! s'exécute si $i \leq -1$

CASE (1, 5:10)

... ! s'exécute si $i=1$ ou $5 \leq i \leq 10$

CASE DEFAULT

... ! s'exécute dans les autres cas

END SELECT

Il faut éviter tout recouvrement entre les différentes possibilités proposées avec l'instruction CASE.

L'instruction GOTO

Pour des questions de lisibilité d'un programme, il faut utiliser cette instruction le moins souvent possible.

Cette instruction redirige le programme vers une autre ligne du code. Cette ligne doit être spécifiée par un numéro.

```
GOTO 10
```

```
...
```

```
10 ... ! On arrive ici avec GOTO 10
```



Les opérations de lecture et d'écriture

Chapitre 3

Ecriture à l'écran

```
PRINT *, "Energie totale = ", Etot, " eV"
```

```
WRITE (*,*) "Energie totale = ", Etot, " eV"
```

PRINT écrit les données au *standard output* (l'écran en général). Le * indique à PRINT de formater automatiquement les données. Les données numériques sont écrites avec toutes leurs décimales représentées en machine.

Pour une écriture formattée, on peut par exemple faire ceci:

```
PRINT '(a,g11.4,a)', "Energie totale = ", Etot, " eV"
```

```
WRITE (*,'(a,g11.4,a)') "Energie totale = ", Etot, " eV"
```

Lecture au clavier

READ *, title, nb, x

READ (*,*) title, nb, x

Le * signifie que la conversion des données vers leur représentation machine doit se faire automatiquement. Les données d'entrée doivent être séparées par des espaces ou des virgules. Pour les chaînes de caractères, le plus simple est de les écrire délimitées par des guillemets.

"Nouveau calcul", 100, 3.

Pour lire à la suite d'une écriture:

PRINT '(a,\$)', "Entrez x : "

READ *, x

Les fichiers: notions générales

Un fichier permet de stocker des données (des "enregistrements") sur le disque. On lui donne généralement une extension .dat, .txt, .out, etc. Il peut être généré par un programme Fortran ou par un éditeur externe comme Notepad.

Un fichier est formaté si les données sont stockées sous forme de caractères alphanumériques. C'est le cas des fichiers produits par un éditeur externe. Il est non formaté si les données sont stockées sous forme binaire (représentation machine).

On peut accéder au contenu d'un fichier de deux manières :

- en accès séquentiel : on doit lire tous les enregistrements qui précèdent celui auquel on souhaite accéder. Les enregistrements peuvent avoir des tailles différentes.
- en accès direct : l'ordre de lecture/écriture indique le numéro de l'enregistrement à traiter. Les enregistrements doivent tous avoir la même taille.

Les fichiers: exemple de base

```
program Main
```

```
real(kind=4) :: x, y
```

```
open(10,file="Fichier.dat")      ! fichier avec écriture
```

```
write(10,*) 1., 2.
```

```
close(10)
```

```
open(10,file="Fichier.dat")      ! fichier avec lecture
```

```
read(10,*) x, y
```

```
close(10)
```

```
print *, x, y
```

```
end program Main
```

Les fichiers: ouverture

Avant de lire et d'écrire dans un fichier, ce dernier doit être ouvert et connecté à une unité logique, c'est-à-dire un numéro compris entre 1 et 2.147.483.647. En général, 5 est préconnecté au clavier et 6 à l'écran. 0 est réservé par le système d'exploitation.

Forme compacte:

```
OPEN(10,file="Fichier.dat")
```

Forme plus complète (avec option):

```
OPEN(UNIT=10,file="Fichier.dat",STATUS="OLD")
```

On ouvre ici le fichier "Fichier.dat" en l'associant à l'unité logique 10. Avec STATUS="OLD", on impose comme condition à cette ouverture de fichier l'existence préalable de Fichier.dat.

Les fichiers: ouverture

Ouverture avec gestion des erreurs:

```
OPEN(UNIT=10,file="Fichier.dat",STATUS="OLD",IOSTAT=ierr)
```

L'option `IOSTAT` permet au programme de continuer en cas d'erreur lors de l'ouverture du fichier. `ierr` doit être déclaré comme entier. Il vaut 0 après l'instruction `OPEN` si l'ouverture du fichier s'est bien déroulée. Il prend une autre valeur sinon.

```
OPEN(UNIT=10,file="Fichier.dat",STATUS="OLD",ERR=100)
```

L'option `ERR` permet également au programme de continuer si une erreur survient lors de l'ouverture du fichier. Dans le cas présent, le programme va à la ligne 100 si une erreur intervient.

Les fichiers: ouverture

Options utiles:

- STATUS="UNKNOWN" (défaut), "OLD", "NEW", "SCRATCH" ou "REPLACE"
- ACTION="READWRITE" (défaut), "READ" ou "WRITE"
- POSITION="ASIS" (défaut), "REWIND" ou "APPEND"
- IOSTAT=ierr
- ERR=label
- FORM="FORMATTED" (défaut) ou "UNFORMATTED"
- ACCESS="SEQUENTIAL" (défaut) ou "DIRECT"
- RECL=n (obligatoire si ACCESS="DIRECT")

Les fichiers: lecture

`READ(10,*) x`

`READ(10,*) x, y`

`READ(UNIT=10,FMT=*) x, y`

`READ(UNIT=10,FMT='(a)') chaine`

UNIT peut être suivi

- du numéro d'un fichier
- d'un astérisque (lecture au clavier)
- d'une chaîne de caractères (pour une conversion de son contenu). Ex: `READ(chaine,*) x`

FMT peut être suivi

- d'un astérisque (format automatique)
- d'une chaîne de caractères spécifiant explicitement le format

Les fichiers: lecture

Lecture avec gestion des erreurs:

```
READ(10,*,iostat=ierr) x
```

En cas d'erreur, ierr prend une valeur différente de 0.

```
READ(10,*,err=100) x
```

En cas d'erreur, le programme va à la ligne 100.

```
READ(10,*,end=20) x
```

Le programme va à la ligne 20 s'il arrive à la fin du fichier.

Les fichiers: écriture

`WRITE(10,*) x`

`WRITE(10,*) x, y`

`WRITE(UNIT=10,FMT=*) x, y`

`WRITE(UNIT=10,FMT='(2g11.4)') x, y`

UNIT peut être suivi

- du numéro d'un fichier
- d'un astérisque (écriture à l'écran)
- d'une chaîne de caractères (pour convertir un nombre en chaîne de caractères). Ex: `WRITE(chaine,*) 5`.

FMT peut être suivi

- d'un astérisque (format automatique)
- d'une chaîne de caractères spécifiant explicitement le format

Les fichiers: écriture

Écriture sans retour à la ligne:

```
WRITE(UNIT=10,FMT='(2g11.4,$)') x, y
```

```
WRITE(UNIT=10,FMT='(2g11.4)',ADVANCE="NO") x, y
```

On ne peut utiliser `ADVANCE="NO"` avec `FMT=*`.

Du texte dans les formats:

```
WRITE(*,'("nmin = ",i3)') nmin
```

Format défini ailleurs dans le programme:

```
WRITE(*,1000) nmin
```

```
1000 FORMAT("nmin = ",i3)
```

Les fichiers: formats

- I4: entier occupant 4 caractères, y compris le signe moins éventuel et des espaces devant.
- F11.4: réel occupant 11 caractères, y compris le signe moins éventuel, le point décimal et des espaces devant, dont 4 chiffres après le point décimal.
- E11.4: réel occupant 11 caractères sous forme mantisse et exposant, dont 4 chiffres après le point décimal.
- G11.4: choix automatique entre F11.4 et E11.4.
- A8: chaîne de caractères de longueur 8.
- L1: valeur logique sous forme T ou F.
- 7X: laisser 7 espaces
- /: aller à la ligne
- \$: rester sur la même ligne
- 3(2I4,3G11.4), (X,A,2I4), etc

Les fichiers: autres opérations

Remonter au début du fichier

`REWIND(10)` ou `REWIND(UNIT=10)`

Remonter d'un enregistrement dans le fichier

`BACKSPACE(10)` ou `BACKSPACE(UNIT=10)`

Les fichiers: fermeture

Forme compacte:

```
CLOSE(10)
```

Forme plus complète (avec option):

```
CLOSE(UNIT=10,STATUS="KEEP")
```

On peut choisir entre STATUS="KEEP" (défaut) ou STATUS="DELETE".



Workspace 'Exemple1': 1 proje

- Exemple1 files
 - Exemple1.f90

FileView

```

Program Main

  real :: x
  integer :: ierr
  character(len=60) :: chaine

! Conversion chaine de caracteres --> nombre

  chaine="3."
  read(chaine,*) x
  print *, "x      = ", x

! Conversion nombre --> chaine de caracteres

  write(chaine,*) 5.
  print *, "chaine = "//chaine
  print *

! Lecture complete d'un fichier

  open(10,file="Exemple1.dat",STATUS="OLD",IOSTAT=ierr,ERR=100)
  if (ierr/=0) then
    print *, "Erreur lors de l'ouverture du fichier (signale par IOSTAT)"
    stop
  endif
  do
    read(10,'(a)',end=20) chaine
    write(*,'(a)') chaine
  enddo
20 close(10)

  stop

100 print *, "Erreur lors de l'ouverture du fichier (signale par ERR)"

end Program Main

```

```

D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre2\Debug\Exemple...
x      = 3.000000
chaine = 5.000000

Maitre Corbeau, sur un arbre perche,
Tenait en son bec un fromage.
Maitre Renard, par l'odeur alleche,
Lui tint a peu pres ce langage :
"He ! bonjour, Monsieur du Corbeau.
Que vous etes joli ! que vous me semblez beau !
Sans mentir, si votre ramage
Se rapporte a votre plumage,
Vous etes le Phenix des hotes de ces bois."
A ces mots le Corbeau ne se sent pas de joie ;
Et pour montrer sa belle voix,
Il ouvre un large bec, laisse tomber sa proie.
Le Renard s'en saisit, et dit : "Mon bon Monsieur,
Apprenez que tout flatteur
Vit aux depens de celui qui l'ecoute :
Cette lecon vaut bien un fromage, sans doute."
Le Corbeau, honteux et confus,
Jura, mais un peu tard, qu'on ne l'y prendrait plus.
Press any key to continue

```

```

-----Configuration: Exemple1 - Win32 Deb
Linking...
Exemple1.exe - 0 error(s), 0 warning(s)
Build Debug Find in Files 1 Find in Files 2

```

Les fichiers: lecture/écriture non formatée

```
OPEN(10,file="Fichier.dat",FORM="UNFORMATTED")
```

```
...
```

```
WRITE(10) x, i
```

```
...
```

```
READ(10) x, i
```

```
...
```

```
CLOSE(10)
```

On peut utiliser des lectures/écritures non formatées pour des fichiers destinés à être relus par l'ordinateur. L'avantage est que les données non formatées prennent moins de place sur le disque, qu'on évite le temps nécessaire à la conversion de la représentation machine vers les données formatées (et l'inverse), et qu'on garde toute la précision des variables numériques.

Les fichiers: utilisation de namelist

Spécification d'un namelist:

```
NAMELIST/nom/variables
```

Lecture/écriture dans un fichier:

```
WRITE(10,NML=nom)
```

```
READ(10,NML=nom)
```

Exemple:

```
NAMELIST/My_NML/x, i
```

```
x = 5.; i = 2
```

```
WRITE(10,NML=My_NML)
```

Le fichier contiendra &My_NML x=5., i=2 /



Workspace 'Exemple4': 1 proje
+ Exemple4 files

```
Program Main  
real :: x  
integer :: i  
  
NAMELIST/My_NML/x, i  
  
x = 5. ; i = 2  
  
open(10,file="Exemple4.dat")  
write(10,NML=My_NML)  
close(10)  
  
open(10,file="Exemple4.dat")  
read(10,NML=My_NML)  
close(10)  
  
print *, x, i  
  
end Program Main
```

Exemple4.dat - Bloc-notes

Fichier Edition Format ?

```
&MY_NML  
X      = 5.000000 ,  
I      =          2  
/
```

"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre1\Debug\Exemple...

```
5.000000      2  
Press any key to continue
```

Linking...

Exemple4.exe - 0 error(s), 0 warning(s)

Les fichiers: lecture/écriture en accès direct

OPEN(10,file="Fichier.dat",ACCESS="DIRECT",RECL=4)

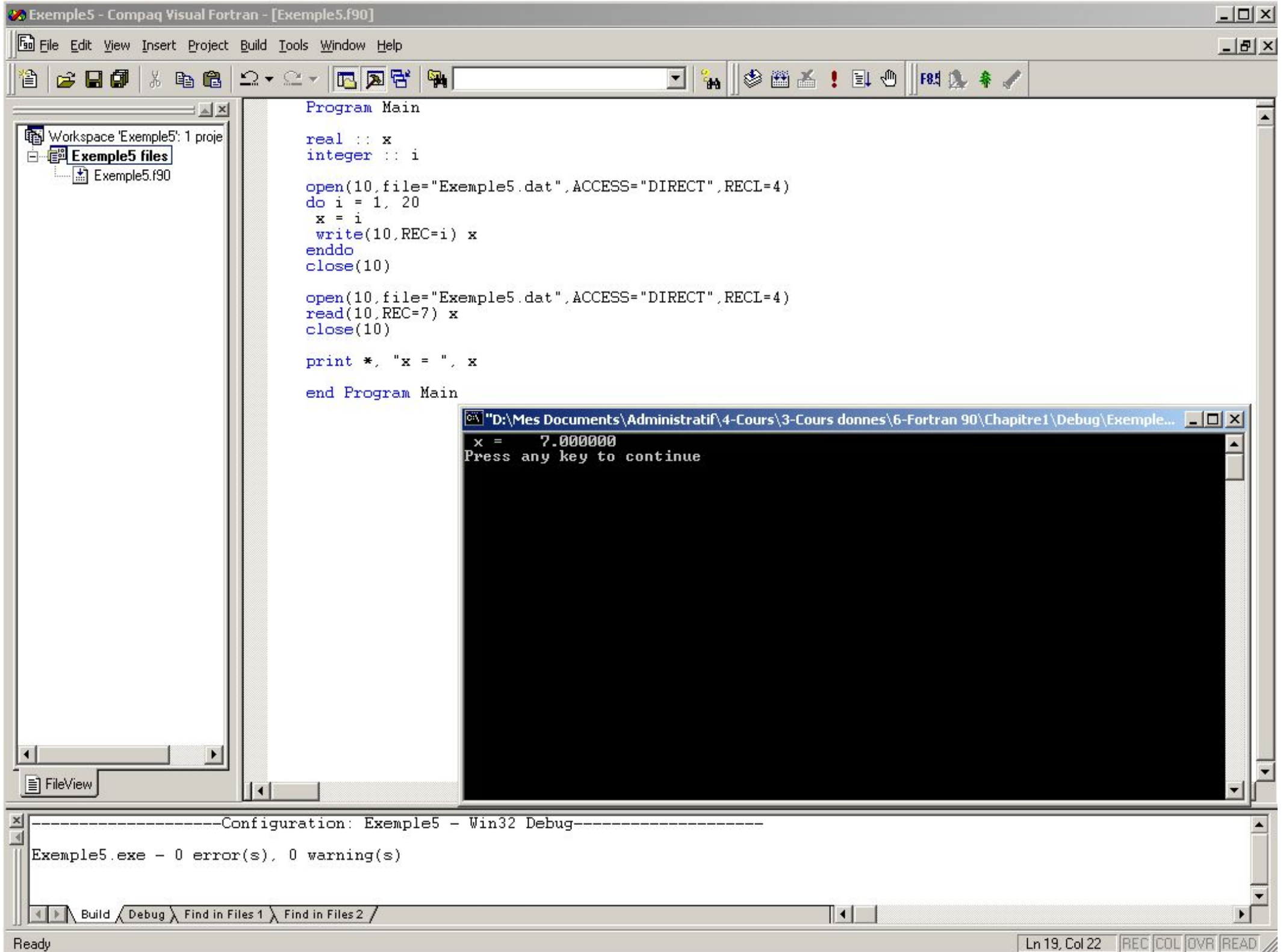
RECL spécifie la longueur en octets de chaque enregistrement (par exemple, 4 pour un réel simple précision).

WRITE(10,REC=n) x

READ(10,REC=n) x

Les READ et WRITE peuvent accéder à chaque enregistrement *directement*. REC spécifie le numéro de l'enregistrement que l'on souhaite lire ou écrire.

CLOSE(10)





Les tableaux

Chapitre 4

Les tableaux: notions générales

Les tableaux englobent les notions de vecteur (tableau 1-D), de matrice (tableau 2-D), de grilles (tableau 3-D), etc.

On peut les générer de manière statique (réservation de la mémoire au moment de la déclaration des variables) ou de manière dynamique (réservation de la mémoire pendant l'exécution du programme).

Ces tableaux sont traités comme des listes de valeurs par certaines fonctions et comme des matrices au sens mathématique par d'autres.

Les tableaux: déclaration statique

```
real :: vecteur(3)
real :: vecteur(1:3)
real,dimension(1:3) :: vecteur
real,dimension(-3:3) :: vecteur

integer,parameter :: n=3
real,dimension(1:n) :: vecteur
real,dimension(1:n,1:n) :: matrice
```

On peut définir des tableaux d'entiers, de réels, de complexes, de variables logiques, de chaînes de caractères, de structures composites, ... bref de tout.

Les tableaux: ordre dans la mémoire

integer,parameter :: n=3
real,dimension(1:n) :: vecteur
real,dimension(1:n,1:n) :: matrice

vecteur(1) matrice(1,1)
vecteur(2) matrice(2,1)
vecteur(3) matrice(3,1)
 matrice(1,2)
 matrice(2,2)
 matrice(3,2)
 matrice(1,3)
 matrice(2,3)
 matrice(3,3)

efficace

```
do j=1,n  
  do i=1,n  
    u=matrice(i,j)...
```

inefficace

```
do i=1,n  
  do j=1,n  
    u=matrice(i,j)...
```

Le premier indice varie le premier en mémoire, ensuite le second, etc.

Les tableaux: déclaration dynamique

L'utilisation dynamique de la mémoire permet de réserver une quantité d'espace mémoire au moment où c'est nécessaire et de la libérer ensuite.

```
integer :: n
```

```
real,dimension(:),allocatable :: vecteur
```

```
real,dimension(:,:),allocatable :: matrice
```

```
n = 3
```

```
allocate (vecteur(1:n),matrice(1:n,1:n))
```

! Allouer les tableaux

```
if (allocated(vecteur)) print *, "vecteur est alloué"
```

! Tester l'allocation

```
if (allocated(matrice)) print *, "matrice est alloué"
```

```
deallocate(vecteur,matrice)
```

! Libérer la mémoire

Les tableaux: déclaration dynamique

Allocation avec gestion des erreurs:

```
integer :: n=3, ierr
```

```
real,dimension(:),allocatable :: vecteur
```

```
allocate (vecteur(1:n),STAT=ierr)
```

```
if (ierr/=0) print *, « Probleme lors de l'allocation »
```

Si la mémoire n'est pas suffisante ou si le tableau est déjà alloué, `ierr` prend une valeur différente de 0.



Workspace 'Exemple1': 1 proje
Exemple1 files
Exemple1.f90

```
Program Main  
  
integer,parameter :: r=8  
integer :: ierr, n  
real(kind=r),allocatable :: vecteur  
  
n = 0 ; ierr = 0  
do while(ierr == 0)  
  n = n + 1024*1024  
  if (allocated(vecteur)) deallocate(vecteur)  
  allocate (vecteur(1:n),stat=ierr)  
enddo  
n = n - 1024*1024  
  
print *, "Memoire disponible (MB) : ", n/(1024*1024)*r  
print *  
  
end Program Main
```

"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre3\Debug\Exemple...
Memoire disponible <MB> : 1640
Press any key to continue_

Linking...
Exemple1.exe - 0 error(s), 0 warning(s)

Les tableaux: remplissage « à la main »

On peut remplir un tableau « à la main » :

Program Main

```
real,dimension(1:3,1:2) :: matrice
```

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

```
matrice(1,1) = 1.
```

```
matrice(2,1) = 2.
```

```
matrice(3,1) = 3.
```

```
matrice(1,2) = 4.
```

```
matrice(2,2) = 5.
```

```
matrice(3,2) = 6.
```

```
end program Main
```

Les tableaux: remplissage via une formule

On peut remplir un tableau via une formule mathématique à condition qu'une telle formule existe et rende compte de chaque élément du tableau.

Program Main

```
real,dimension(1:3,1:2) :: matrice  
integer :: i, j
```

```
do j = 1, 2  
  do i = 1, 3  
    matrice(i,j) = i + (j-1) * 3  
  enddo  
enddo
```

```
end program Main
```

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

Les tableaux: remplissage à partir d'un fichier

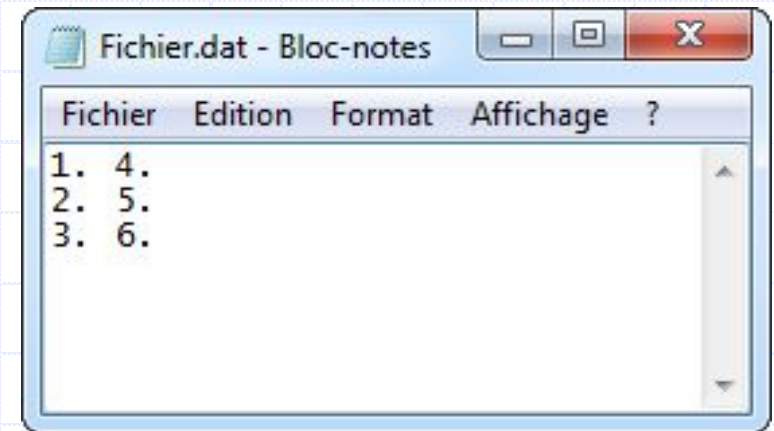
On peut remplir un tableau en lisant le contenu d'un fichier.

Program Main

```
real,dimension(1:3,1:2) :: matrice
integer :: i, j

open(10,file=« Fichier.dat »)
do i = 1, 3
  read(10,*) matrice(i,1), matrice(i,2)
enddo
close(10)

end program Main
```



Les tableaux: remplissage à partir d'un fichier

On peut remplir un tableau en lisant le contenu d'un fichier.

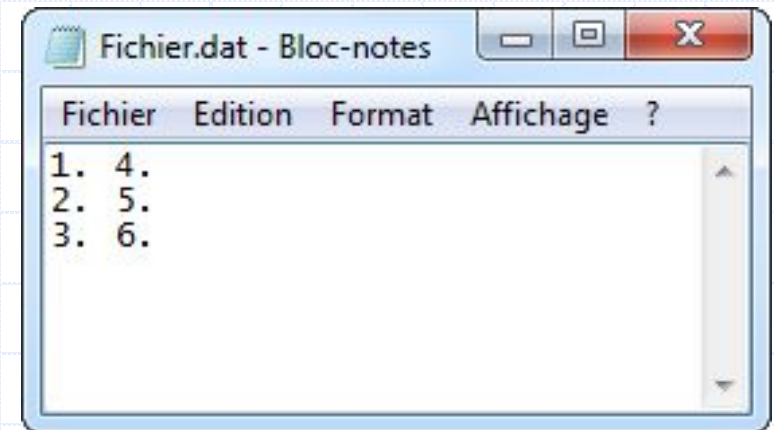
Program Main

```
real,dimension(1:3,1:2) :: matrice  
integer :: i, j
```

```
open(10,file=« Fichier.dat »)  
do i = 1, 3  
  read(10,*) (matrice(i,j), j=1,2)  
enddo  
close(10)
```

end program Main

La boucle sur j se fait ici via un itérateur.



Les tableaux: array constructor

On peut définir le contenu d'un tableau à une dimension en utilisant un *array constructor*.

```
integer :: i
```

```
real,dimension(1:5) :: vecteur
```

```
vecteur = (/ 2., 4., 6., 8., 10. /)      ! donne 2., 4., 6., 8., 10.
```

```
vecteur = (/ (2.*i, i=1, 5) /)
```

```
vecteur = (/ 2., (2.*i, i=2, 4), 10. /)
```

```
vecteur = (/ (1.*i, i=5, 1, -1) /)      ! donne 5., 4., 3., 2., 1.
```

On peut également utiliser un *array constructor* au moment de la déclaration.

```
real,dimension(1:5) :: vecteur = (/ 2., 4., 6., 8., 10. /)
```


Les tableaux: la fonction RESHAPE

Pour un tableau à plusieurs dimensions, on peut utiliser la fonction RESHAPE.

```
real,dimension(1:3,1:2) :: matrice
```

```
matrice=reshape( (/1.,2.,3.,4.,5.,6./) , (/3,2/) )
```

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

Le premier argument contient la liste des valeurs à utiliser.
Le second argument donne la forme de la matrice.

Les tableaux: la fonction RESHAPE

On peut modifier l'ordre selon lequel la matrice est remplie.

```
real,dimension(1:3,1:2) :: matrice
```

```
matrice=reshape( (/1.,2.,3.,4.,5.,6./) , (/3,2/), order=(/2,1/) )
```

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

On force ici un remplissage dans lequel le deuxième argument de matrice varie en premier et le premier argument en second.

Les tableaux: la fonction RESHAPE

On peut préciser les valeurs « par défaut » à utiliser pour le remplissage, pour le cas où le premier argument ne spécifie pas tous les éléments de la matrice.

```
real,dimension(1:3,1:2) :: matrice
```

```
matrice=reshape( (/1.,2.,3./) , (/3,2/), order=(/2,1/) , pad=(/0./) )
```

$$\text{matrice} = \begin{pmatrix} 1. & 2. \\ 3. & 0. \\ 0. & 0. \end{pmatrix}$$

```
matrice=reshape( (/1.,2.,3./) , (/3,2/), order=(/2,1/) , pad=(/0.,1./) )
```

$$\text{matrice} = \begin{pmatrix} 1. & 2. \\ 3. & 0. \\ 1. & 0. \end{pmatrix}$$

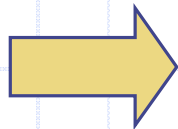
Les tableaux: affectations globales

`real,dimension(1:3,1:2) :: matrice1, matrice2`

`matrice1(1,1) = 0.` ! affectation d'un seul élément

`matrice1 = 0.` ! affectation de toute la matrice

`matrice1 = matrice2` ! copie du contenu de matrice2 dans matrice1



`matrice1(1,1) = matrice2(1,1)`
`matrice1(2,1) = matrice2(2,1)`
`matrice1(3,1) = matrice2(3,1)`
`matrice1(1,2) = matrice2(1,2)`
`matrice1(2,2) = matrice2(2,2)`
`matrice1(3,2) = matrice2(3,3)`

Pour des affectations globales comme `matrice1=matrice2`, il est nécessaire que les deux matrices aient **la même forme**. Pas nécessairement les mêmes bornes.

Les tableaux: affectations globales

Ce qui suit marche aussi :

```
real,dimension(1:3,1:2) :: matrice1  
real,dimension(-1:1,0:1) :: matrice2
```

`matrice1 = matrice2` ! copie du contenu de `matrice2` dans `matrice1`

Le résultat est le suivant :

```
matrice1(1,1) = matrice2(-1,0)  
matrice1(2,1) = matrice2(0,0)  
matrice1(3,1) = matrice2(1,0)  
matrice1(1,2) = matrice2(-1,1)  
matrice1(2,2) = matrice2(0,1)  
matrice1(3,2) = matrice2(1,1)
```

Les tableaux: sections

On peut adresser des sections de tableaux en utilisant un indiçage du type `<début>:<fin>:<incrément>`.

```
real,dimension(1:3) :: vecteur
```

```
print *, vecteur(1:3:2)      ! donne (/ vecteur(1), vecteur(3) /)
```

Si `<début>` est omis, il vaut la borne inférieure de l'indice.

Si `<fin>` est omis, il vaut la borne supérieure de l'indice.

Si `<incrément>` est omis, il vaut 1. Ces arguments peuvent prendre des valeurs négatives.

Les tableaux: sections

real,dimension(1:3) :: vecteur
real,dimension(1:3,1:3) :: matrice

```
print *, vecteur(1:2)      ! les 2 premiers éléments de vecteur
print *, matrice(1:1,1:3) ! la première ligne de matrice
print *, matrice(1:3:2,1:3) ! la première et la troisième ligne de matrice
print *, matrice(3:1:-2,1:3) ! la troisième et la première ligne de matrice
print *, matrice(:,2)     ! le premier bloc (2x2) de matrice
print *, matrice(:,1)    ! la première colonne de matrice
print *, matrice(:,2,1)  ! 1 élément sur 2 de la première colonne
                        ! de matrice
```

Il faut noter que `matrice(:,1)` devient un tableau de rang 1 (l'équivalent d'un vecteur). Spécifier une constante comme indice a donc pour effet de réduire le rang du tableau considéré.

Les tableaux: vector subscripts

On peut adresser les éléments d'un tableau en utilisant comme indice un vecteur d'entiers.

```
integer,dimension(1:3) :: subscript = (/ 1, 4, 7 /)
```

```
integer,dimension(1:10) :: permutation = (/2, 3, 4, 5, 6, 7, 8, 9, 10, 1/)
```

```
real,dimension(1:10,1:10) :: matrice
```

```
print *, matrice(subscript, 5)           ! éléments (1,5), (4,5) et (7,5)
```

```
print *, matrice(subscript, subscript) ! éléments (1,1), (4,1) , (7,1),  
                                         ! (1,4), (4,4) , (7,4), (1,7), (4,7) et (7,7)  
                                         ! dans une sous-matrice (3x3)
```

```
matrice(1:10, 1) = matrice(permutation, 1) ! permutation de la  
                                             ! première colonne
```

```
matrice(subscript, 1) = matrice(1:3, 2)   ! spécification de certains
```

```
matrice(subscript, 1) = (/ 1., 2., 3. /)  ! éléments (subscript doit
```

```
matrice(subscript, 1) = 0.                ! contenir ici des valeurs
```

```
! toutes différentes)
```


Les tableaux: expressions

Les expressions entre tableaux doivent impliquer des tableaux de même forme.

`real,dimension(1:3) :: vecteur`

`real,dimension(1:3,1:3) :: matrice1`

`real,dimension(0:2,-1:1) :: matrice2`

`matrice1 = matrice1 * matrice2` ! Il s'agit du produit entre éléments
! pris individuellement, pas du produit
! matriciel !!!

`matrice1 = exp(matrice2)` ! Ces opérations sont réalisées en

`matrice1 = cos(matrice2)` ! prenant chaque élément

`matrice1 = sin(matrice2)` ! individuellement. Il ne s'agit par
! exemple pas de l'exponentielle de
! matrice !

`matrice1(:,1) = matrice2(0,:) + vecteur(:)` ! Opérations permises car

`matrice1(:,1) = sqrt(vecteur(:))` ! formes identiques.

En Fortran :

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11} * b_{11} & a_{12} * b_{12} \\ a_{21} * b_{21} & a_{22} * b_{22} \end{pmatrix}$$

$$\text{exp} \left(\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \right) = \begin{pmatrix} \text{exp}(a_{11}) & \text{exp}(a_{12}) \\ \text{exp}(a_{21}) & \text{exp}(a_{22}) \end{pmatrix}$$

$$\text{cos} \left(\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \right) = \begin{pmatrix} \text{cos}(a_{11}) & \text{cos}(a_{12}) \\ \text{cos}(a_{21}) & \text{cos}(a_{22}) \end{pmatrix}$$

etc

Les tableaux: l'instruction WHERE

Dans une affectation d'un tableau, il est possible de soumettre l'exécution de cette affectation à une condition qui sera testée pour chaque élément du tableau.

```
real,dimension(1:2,1:2) :: matrice
```

```
WHERE (matrice > 0.) matrice = log10(matrice)
```

```
WHERE (matrice > 0.)
```

$$\begin{pmatrix} 100. & 10. \\ 0.1 & 0. \end{pmatrix} \rightarrow \begin{pmatrix} 2. & 1. \\ -1. & 0. \end{pmatrix}$$

```
    matrice = log10(matrice)
```

```
ELSEWHERE
```

```
    matrice = -100.
```

$$\begin{pmatrix} 100. & 10. \\ 0.1 & 0. \end{pmatrix} \rightarrow \begin{pmatrix} 2. & 1. \\ -1. & -100. \end{pmatrix}$$

```
END WHERE
```

Il est important que le masque conditionnant le WHERE ainsi que les tableaux concernés aient tous la même forme.

Les tableaux: l'instruction WHERE

On peut en réalité utiliser directement un masque.
Les choses se passent alors comme suit :

```
real,dimension(1:2,1:2) :: matrice  
logical,dimension(1:2,1:2) :: mask
```

```
mask = reshape( (/ .true., .true., .true., .false. /) , (/ 2, 2 /) )
```

```
WHERE (mask) matrice = log10(matrice)
```

$$\begin{pmatrix} 100. & 10. \\ 0.1 & 0. \end{pmatrix} \rightarrow \begin{pmatrix} 2. & 1. \\ -1. & 0. \end{pmatrix}$$

Il faut que tous les objets concernés (mask et matrice) aient à nouveau la même forme.

Les tableaux: fonctions intrinsèques

```
real,dimension(1:3,1:3) :: matrice
```

$$\text{matrice} = \begin{pmatrix} 1. & 4. & 7. \\ 2. & 5. & 8. \\ 3. & 6. & 9. \end{pmatrix}$$

```
matrice=reshape((/(1.*i,i=1,9)/),(/3,3/))
```

```
print *, LBOUND(matrice)           ! donne (/ 1, 1 /)
```

```
print *, LBOUND(matrice,DIM=1)     ! donne 1
```

```
print *, UBOUND(matrice)           ! donne (/ 3, 3 /)
```

```
print *, UBOUND(matrice,DIM=1)     ! donne 3
```

```
print *, SIZE(matrice)              ! donne 9
```

```
print *, SIZE(matrice,DIM=1)       ! donne 3
```

```
print *, SHAPE(matrice)             ! donne (/ 3, 3 /)
```

Les tableaux: MINVAL et MAXVAL

```
real,dimension(1:3,1:3) :: matrice
```

$$\text{matrice} = \begin{pmatrix} 1. & 4. & 7. \\ 2. & 5. & 8. \\ 3. & 6. & 9. \end{pmatrix}$$

```
matrice=reshape((/(1.*i,i=1,9)/),(/3,3/))
```

```
print *, MINVAL(matrice)           ! donne 1.
print *, MINVAL(matrice,DIM=1)     ! donne (/ 1., 4., 7. /)
print *, MINVAL(matrice,DIM=2)     ! donne (/ 1., 2., 3. /)
```

```
print *, MAXVAL(matrice)           ! donne 9.
print *, MAXVAL(matrice,DIM=1)     ! donne (/ 3., 6., 9. /)
print *, MAXVAL(matrice,DIM=2)     ! donne (/ 7., 8., 9. /)
```

```
print *, MINVAL(matrice,matrice>=5.) ! donne 5.
```

DIM indique la dimension selon laquelle l'instruction est réalisée.

```
print *, MINLOC(matrice)           ! donne (/ 1, 1 /)
print *, MAXLOC(matrice)           ! donne (/ 3, 3 /)
```

Les tableaux: SUM et PRODUCT

```
real,dimension(1:3,1:3) :: matrice
```

$$\text{matrice} = \begin{pmatrix} 1. & 4. & 7. \\ 2. & 5. & 8. \\ 3. & 6. & 9. \end{pmatrix}$$

```
matrice=reshape((/(1.*i,i=1,9)/),(/3,3/))
```

```
print *, SUM(matrice)           ! donne 45.
```

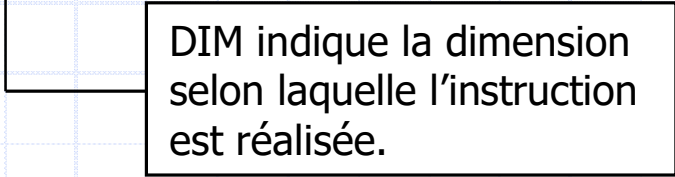
```
print *, SUM(matrice,DIM=1)     ! donne (/ 6., 15., 24. /)
```

```
print *, SUM(matrice,DIM=2)     ! donne (/ 12., 15., 18. /)
```

```
print *, PRODUCT(matrice)       ! donne 362880.
```

```
print *, PRODUCT(matrice,DIM=1) ! donne (/ 6., 120., 504. /)
```

```
print *, PRODUCT(matrice,DIM=2) ! donne (/ 28., 80., 162. /)
```



DIM indique la dimension
selon laquelle l'instruction
est réalisée.

Les tableaux: opérations matricielles

$$\text{vecteur} = \begin{pmatrix} 1. \\ 2. \\ 3. \end{pmatrix}$$

```
real,dimension(1:3) :: vecteur  
real,dimension(1:3,1:3) :: matrice
```

$$\text{matrice} = \begin{pmatrix} 1. & 4. & 7. \\ 2. & 5. & 8. \\ 3. & 6. & 9. \end{pmatrix}$$

```
vecteur=(/ 1., 2., 3. /)  
matrice=reshape((/(1.*i,i=1,9)/),(/3,3/))
```

```
print *, MATMUL(matrice, matrice)      ! produit matriciel
```

$$\begin{pmatrix} 1. & 4. & 7. \\ 2. & 5. & 8. \\ 3. & 6. & 9. \end{pmatrix} \begin{pmatrix} 1. & 4. & 7. \\ 2. & 5. & 8. \\ 3. & 6. & 9. \end{pmatrix} = \begin{pmatrix} 30. & 66. & 102. \\ 36. & 81. & 126. \\ 42. & 96. & 150. \end{pmatrix}$$

```
print *, MATMUL(matrice, vecteur)      ! produit matrice-vecteur
```

$$\begin{pmatrix} 1. & 4. & 7. \\ 2. & 5. & 8. \\ 3. & 6. & 9. \end{pmatrix} \begin{pmatrix} 1. \\ 2. \\ 3. \end{pmatrix} = \begin{pmatrix} 30. \\ 36. \\ 42. \end{pmatrix}$$

Les tableaux: opérations matricielles

$$\text{vecteur} = \begin{pmatrix} 1. \\ 2. \\ 3. \end{pmatrix}$$

```
real,dimension(1:3) :: vecteur  
real,dimension(1:3,1:3) :: matrice
```

$$\text{matrice} = \begin{pmatrix} 1. & 4. & 7. \\ 2. & 5. & 8. \\ 3. & 6. & 9. \end{pmatrix}$$

```
vecteur=(/ 1., 2., 3. /)  
matrice=reshape((/(1.*i,i=1,9)/),(/3,3/))
```

```
print *, DOT_PRODUCT(vecteur, vecteur)      ! produit scalaire
```

$$(1. \quad 2. \quad 3.) \begin{pmatrix} 1. \\ 2. \\ 3. \end{pmatrix} = 14.$$

```
print *, TRANSPOSE(matrice)                ! transposée
```

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

Les tableaux: ALL, ANY et COUNT

```
real,dimension(1:3,1:3) :: matrice
```

$$\text{matrice} = \begin{pmatrix} 1. & 4. & 7. \\ 2. & 5. & 8. \\ 3. & 6. & 9. \end{pmatrix}$$

```
matrice=reshape((/(1.*i,i=1,9)/),(/3,3/))
```

```
if (ALL(matrice >= 0.)) print *, « Tous les éléments sont positifs. »
```

```
if (ANY(matrice >= 0.)) print *, « Au moins un élément est positif. »
```

```
print *, COUNT(matrice >= 0.)           ! donne 9
```

```
print *, ALL(matrice>=5.,DIM=1)         ! donne (/ F, F, T /)
```

```
print *, ALL(matrice>=5.,DIM=2)         ! donne (/ F, F, F /)
```

```
print *, ANY(matrice>=5.,DIM=1)         ! donne (/ F, T, T /)
```

```
print *, ANY(matrice>=5.,DIM=2)         ! donne (/ T, T, T /)
```

DIM indique la dimension
selon laquelle l'instruction
est réalisée.

Les tableaux: la fonction PACK

```
real,dimension(1:3,1:3) :: matrice
```

$$\text{matrice} = \begin{pmatrix} 1. & 4. & 7. \\ 2. & 5. & 8. \\ 3. & 6. & 9. \end{pmatrix}$$

```
matrice=reshape((/(1.*i,i=1,9)/),(/3,3/))
```

```
print *, PACK(matrice, matrice >= 5.) ! donne (/ 5., 6., 7., 8., 9. /)
```

```
print *, PACK(matrice, matrice >= 5., (/1./)) ! donne (/ 5. /)
```

```
print *, PACK(matrice, matrice >= 5., (/1.,1./)) ! donne (/ 5., 6. /)
```

masque

décide la longueur
du résultat

Les tableaux: changements de forme

real,dimension(1:4) :: vecteur
real,dimension(1:2,1:2) :: matrice

$$\text{matrice} = \begin{pmatrix} 1. & 3. \\ 2. & 4. \end{pmatrix}$$

matrice=reshape((/(1.*i,i=1,4)/),(/2,2/))

vecteur = reshape(matrice, (/4/))

$$\begin{pmatrix} 1. & 3. \\ 2. & 4. \end{pmatrix} \rightarrow \begin{pmatrix} 1. \\ 2. \\ 3. \\ 4. \end{pmatrix}$$

matrice = reshape(vecteur, (/2,2/))

$$\begin{pmatrix} 1. \\ 2. \\ 3. \\ 4. \end{pmatrix} \rightarrow \begin{pmatrix} 1. & 3. \\ 2. & 4. \end{pmatrix}$$



Les procédures

Chapitre 5

Structure d'un programme

Un code Fortran est composé en général des quatre **unités de programme** suivantes:

- PROGRAM: le programme principal
- SUBROUTINE: les sous-routines (internes ou externes)
- FUNCTION: les fonctions (internes ou externes)
- MODULE: des modules pouvant reprendre des instructions d'affectation, des sous-routines, des fonctions ou les interfaces nécessaires à leur utilisation.

Le programme principal, les sous-routines et les fonctions portent aussi le nom de **procédure**. Une procédure est **interne** si elle est définie à l'intérieur d'une unité de programme. Cette unité de programme peut être (i) une "procédure hôte" faisant appel à cette procédure interne ou (ii) un module dans le cas d'une procédure de module. Une procédure est **externe** si elle n'est contenue dans aucune autre unité de programme.

Procédures internes ou externes

PROGRAM Main

...

! On trouve ici les instructions de spécification
! et les instructions exécutables du programme
! principal.

CONTAINS

...

! On définit ici les **procédures internes**.
! Elles ont accès à toutes les variables
! du programme principal (sauf si utilisation
! de noms identiques).

END PROGRAM Main

! On définit ici les **procédures externes**.
! Elles sont complètement indépendantes
! du programme principal et n'ont a priori
! rien en commun.

Procédures internes ou externes

```
PROGRAM Main
integer :: i=5 ←
call SousRoutine
stop
CONTAINS
SUBROUTINE SousRoutine
integer :: j=3 ←
print *, i, j
END SUBROUTINE SousRoutine
END PROGRAM Main
```

i est une variable du programme principal.
Les procédures internes au programme principal y ont accès (sauf si utilisation d'un nom identique).
On parle de « variable globale ».

j est une variable de la procédure interne.
Le programme principal n'y a pas accès.
On parle de « variable locale ».

53

Les procédures internes ont accès à toutes les variables de la « procédure hôte » (ici le programme principal), sauf celles qui sont déclarées avec un nom identique dans la procédure interne. Ce nom réfère alors aux variables locales.

Procédures internes ou externes

```
PROGRAM Main
integer :: i=5
call SousRoutine
stop
CONTAINS
SUBROUTINE SousRoutine
integer :: i=3
print *, i
end SUBROUTINE SousRoutine
END PROGRAM Main
```

i est une variable du programme principal. Les procédures internes au programme principal y ont accès (sauf si utilisation d'un nom identique).

i est une variable de la procédure interne. Ce i est indépendant du i du programme principal.

C'est le « i local » qui est imprimé.

3

Une procédure ainsi que ses procédures internes sont compilées ensemble. Le compilateur peut décider de remplacer l'appel à la procédure interne par les instructions contenues dans celle-ci (in-lining).

Les sous-routines: syntaxe générale

SUBROUTINE <nom> (<dummy arguments>)

- ! Instructions de spécification des dummy arguments
- ! Instructions de spécification des objets locaux
- ! Instructions exécutables

END SUBROUTINE <nom>

Une sous-routine peut contenir des procédures internes (à condition qu'il ne s'agisse pas déjà d'une procédure interne). Il suffit d'utiliser CONTAINS comme précédemment. Ces procédures internes ont alors accès aux dummy arguments de la sous-routine ainsi qu'aux variables définies dans celle-ci (sauf si utilisation de noms identiques).

Les dummy arguments peuvent être des variables, des tableaux, des pointeurs, des noms de procédures internes ou externes, ou des noms de procédures de module.

Les sous-routines: l'attribut INTENT

Les dummy arguments qui sont des variables ou des tableaux doivent être déclarés avec l'attribut INTENT(IN), INTENT(OUT) ou INTENT(INOUT).

INTENT(IN) : variables dont la valeur à l'entrée de la sous-routine est utilisée. Cette valeur est fournie par la procédure qui appelle cette sous-routine. Ces variables ne peuvent être modifiées par la sous-routine.

INTENT(OUT) : variables dont la valeur éventuelle à l'entrée de la sous-routine n'est pas utilisée, mais qui sont modifiées par la sous-routine.

INTENT(INOUT) : variables dont la valeur à l'entrée de la sous-routine est utilisée et qui sortent avec une valeur différente.

Les dummy arguments qui sont des pointeurs doivent être déclarés avec l'attribut POINTER. Les dummy arguments qui sont des noms de procédures doivent être déclarés par une interface (voir plus loin).

Les sous-routines: exemple

SUBROUTINE SousRoutine (x, y, z)

real,intent(in) :: x ! On utilise la valeur de x sans changer x

real,intent(out) :: y ! On n'utilise pas la valeur de y, mais on change y

real,intent(inout) :: z ! On utilise la valeur de z et on change z

y = 2. * x

z = z + y

END SUBROUTINE SousRoutine

On peut sortir à n'importe quel moment de la sous-routine grâce à l'instruction RETURN.

Appel d'une sous-routine avec CALL

On appelle une sous-routine par l'instruction CALL.

```
CALL <nom> (<actual arguments>)
```

Il y a autant de <actual arguments> que de <dummy arguments> dans l'instruction SUBROUTINE. Dans le cas de variables, les actual et dummy arguments doivent avoir le même type et le même kind. Ils peuvent porter des noms différents.

Il faut par ailleurs inclure dans la procédure appelante l'interface de la sous-routine (sauf s'il s'agit d'une sous-routine interne). Cette interface ne doit reprendre que la spécification des dummy arguments.

Implémentation d'une sous-routine externe

PROGRAM Main

real :: x, y, z

INTERFACE

SUBROUTINE SousRoutine (x, y, z)

real,intent(in) :: x ! On utilise la valeur de x sans changer x

real,intent(out) :: y ! On n'utilise pas la valeur de y, mais on change y

real,intent(inout) :: z ! On utilise la valeur de z et on change z

END SUBROUTINE SousRoutine

END INTERFACE

x = 1. ; z = 2.

CALL SousRoutine (x, y, z)

print *, x, y, z

END PROGRAM Main

L'interface sert à décrire les dummy arguments de la sous-routine. Le compilateur a besoin de cette information au moment de compiler le programme principal.

Le nom des actual arguments ne doit pas nécessairement être le même que celui des dummy arguments.

Implémentation d'une sous-routine externe

PROGRAM Main

real :: my_x, my_y, my_z

L'interface sert à décrire les dummy arguments de la sous-routine. Le compilateur a besoin de cette information au moment de compiler le programme principal.

INTERFACE

SUBROUTINE SousRoutine (x, y, z)

real,intent(in) :: x ! On utilise la valeur de x sans changer x

real,intent(out) :: y ! On n'utilise pas la valeur de y, mais on change y

real,intent(inout) :: z ! On utilise la valeur de z et on change z

END SUBROUTINE SousRoutine

END INTERFACE

my_x = 1. ; my_z = 2.

CALL SousRoutine (my_x, my_y, my_z)

print *, my_x, my_y, my_z

Le nom des actual arguments ne doit pas nécessairement être le même que celui des dummy arguments.

END PROGRAM Main

Exemple1 - Compaq Visual Fortran - [Exemple1.f90]

File Edit View Insert Project Build Tools Window Help

Workspace 'Exemple1': 1 proje
Exemple1 files
Exemple1.f90

```
PROGRAM Main
  real :: x, y, z

  INTERFACE
    SUBROUTINE SousRoutine (x, y, z)
      real,intent(in) :: x
      real,intent(out) :: y
      real,intent(inout) :: z
    END SUBROUTINE SousRoutine
  END INTERFACE

  x = 1. ; z = 2.
  CALL SousRoutine (x,y,z)
  print *, x, y, z

END PROGRAM Main

SUBROUTINE SousRoutine (x, y, z)

  real,intent(in) :: x
  real,intent(out) :: y
  real,intent(inout) :: z

  y = 2. * x
  z = z + y

END SUBROUTINE SousRoutine
```

"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre4\Debug\Exemple...
1.000000 2.000000 4.000000
Press any key to continue

FileView

Linking...
Exemple1.exe - 0 error(s), 0 warning(s)

Build Debug Find in Files 1 Find in Files 2

Implémentation d'une sous-routine interne

```
PROGRAM Main
  real :: my_x, my_y, my_z
  my_x = 1. ; my_z = 2.
  CALL SousRoutine (my_x, my_y, my_z)
  print *, my_x, my_y, my_z
CONTAINS
  SUBROUTINE SousRoutine (x, y, z)
    real,intent(in) :: x
    real,intent(out) :: y
    real,intent(inout) :: z
    y = 2. * x
    z = z + y
  END SUBROUTINE SousRoutine
END PROGRAM Main
```

Une interface n'est pas nécessaire car la sous-routine est interne au programme.

Faut-il construire des sous-routines internes ou externes ?

Les avantages d'une sous-routine interne (par rapport à une sous-routine externe):

- Il n'est pas nécessaire de construire une interface.
- La sous-routine interne a accès à toutes les variables de la procédure hôte (sauf si utilisation de noms identiques). Il n'est donc pas nécessaire de les passer par arguments ou via des modules (voir plus loin).

Les désavantages d'une sous-routine internes:

- Il y a un risque important de modifier accidentellement les variables de la procédure hôte.
- Une sous-routine interne n'est accessible que par la procédure qui la contient. Pour une utilisation par plusieurs procédures, il faut absolument créer des sous-routines externes.

Je recommande de construire a priori des sous-routines externes, sauf s'il y a un avantage indéniable à faire autrement.

Utilisation plus avancée du CALL

On peut utiliser CALL en mettant simplement les actual arguments dans le même ordre que les dummy arguments:

```
CALL SousRoutine (3., my_y, my_z)
```

On parle de « **positional arguments** ».

Une autre façon de procéder consiste à rappeler le nom des dummy arguments dans l'instruction CALL:

```
CALL SousRoutine (x=3., y=my_y, z=my_z)
```

actual arguments



nom des dummy arguments

L'avantage est qu'on peut modifier l'ordre des arguments.
On parle de « **keywords arguments** ».

Utilisation plus avancée du CALL

On peut mélanger la méthode « positional arguments » avec celle des « keyword arguments ». Le principe est que dès qu'un argument est donné avec son keyword, les arguments suivants doivent l'être également.

```
CALL SousRoutine (3., y=my_y, z=my_z)
```

L'attribut SAVE

Si l'on souhaite qu'une variable locale conserve sa valeur d'un appel à l'autre de la sous-routine, on peut lui donner l'attribut SAVE. Certains compilateurs le font automatiquement.

```
INTEGER, SAVE :: i
```

On peut par ailleurs, au moment de la spécification de cette variable locale, lui donner une valeur initiale.

```
INTEGER, SAVE :: i = 0
```

Cette valeur initiale est utilisée lors du premier appel de la sous-routine. Aux appels suivants, la valeur de cette variable peut avoir changé.

Finalement, l'instruction SAVE confère à toutes les variables locales l'attribut SAVE.

Exemple2 - Compaq Visual Fortran - [Exemple2.f90]

File Edit View Insert Project Build Tools Window Help

Workspace 'Exemple2': 1 proje
Exemple2 files
Exemple2.f90

```
PROGRAM Main
  integer :: n

  INTERFACE
    SUBROUTINE SousRoutine
    END SUBROUTINE SousRoutine
  END INTERFACE

  do n = 1, 10
    call SousRoutine
  enddo

END PROGRAM Main

SUBROUTINE SousRoutine
  integer,save :: i = 0

  i = i + 1
  print *, i
END SUBROUTINE SousRoutine
```

"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre5\Debug\Exemple..."

```
1
2
3
4
5
6
7
8
9
10
Press any key to continue_
```

FileView

Linking...

Exemple2.exe - 0 error(s), 0 warning(s)

Build Debug Find in Files 1 Find in Files 2

Les fonctions

Une fonction est une procédure appelée à l'intérieur d'une expression et dont le résultat est utilisé dans cette expression.

```
...  
d = Somme (a, b, c) * 2.  
...
```

La fonction Somme est définie de la manière suivante:

```
FUNCTION Somme (x, y, z)  
  real :: Somme  
  real, intent(in) :: x, y, z  
  Somme = x + y + z  
END FUNCTION Somme
```

Les fonctions

Les dummy arguments d'une fonction doivent être déclarés comme ceux d'une sous-routine. Ces arguments ont généralement l'attribut `INTENT(IN)`. Il est nécessaire par ailleurs de spécifier le type et le kind de la fonction.

Comme pour les sous-routines, une fonction peut être interne ou externe. Dans le cas d'une fonction externe, il est nécessaire de construire une interface. Cette interface reprend les instructions de spécification de la fonction et des dummy arguments.

Finalement, il est possible de sortir à tout moment d'une fonction grâce à l'instruction `RETURN`.

Implémentation d'une fonction externe

PROGRAM Main

real :: a, b, c, d

INTERFACE

FUNCTION Somme (x, y, z)

real :: Somme

real, intent(in) :: x, y, z

END FUNCTION Somme

END INTERFACE

a = 1. ; b = 2. ; c = 3.

d = Somme (a, b, c) * 2.

print *, d

END PROGRAM Main

L'interface reprend les instructions de spécification de la fonction et des dummy arguments. Le compilateur a besoin de cette information au moment de compiler le programme principal.



Workspace 'Exemple3': 1 proje

- Exemple3 files
 - Exemple3.f90

FileView

```

PROGRAM Main
  real :: a, b, c, d

  INTERFACE
    FUNCTION Somme (x, y, z)
      real :: Somme
      real, intent(in) :: x, y, z
    END FUNCTION Somme
  END INTERFACE

  a = 1. ; b = 2. ; c = 3.
  d = Somme (a, b, c) * 2.
  print *, d

END PROGRAM Main

FUNCTION Somme (x, y, z)

  real :: Somme
  real, intent(in) :: x, y, z

  Somme = x + y + z

END FUNCTION Somme

```

"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre5\Debug\Exemple...

```

12.00000
Press any key to continue_

```

Implémentation d'une fonction interne

PROGRAM Main

```
real :: a, b, c, d
```

```
a = 1. ; b = 2. ; c = 3.
```

```
d = Somme (a, b, c) * 2.
```

```
print *, d
```

Une interface n'est pas nécessaire car la fonction est interne au programme.

CONTAINS

```
FUNCTION Somme (x, y, z)
```

```
real :: Somme
```

```
real, intent(in) :: x, y, z
```

```
Somme = x + y + z
```

```
END FUNCTION Somme
```

END PROGRAM Main

Les interfaces

Nous avons vu qu'une interface consiste en un bloc d'instructions que l'on place parmi les instructions de spécification d'un programme ou d'une procédure qui fait appel à une procédure externe.

Une interface est composée

- de l'instruction de la procédure (SUBROUTINE ou FUNCTION)
- des instructions de spécification de la fonction
- des instructions de spécification des dummy arguments
- de END SUBROUTINE ou END FUNCTION

Nous verrons plus tard que toutes les interfaces utilisées dans un projet peuvent être rassemblées dans un **module**. Il suffira alors de faire référence à ce module pour inclure dans une procédure faisant appel à une procédure externe les interfaces nécessaires.



Les procédures: notions avancées

Chapitre 6

Tableaux de dimensions fixes comme arguments

La première manière d'utiliser des tableaux comme arguments d'une procédure consiste à donner leurs dimensions explicitement.

```
FUNCTION PRODUIT_SCALAIRE ( vec1 , vec2 )  
  REAL :: PRODUIT_SCALAIRE  
  REAL,DIMENSION(1:3),INTENT(IN) :: vec1, vec2  
  
  PRODUIT_SCALAIRE = SUM ( vec1 * vec2 )  
END FUNCTION PRODUIT_SCALAIRE
```

La procédure qui appelle cette fonction doit fournir comme actual arguments des tableaux ou des sections de tableaux ayant le même type, le même kind et la même forme que les dummy arguments de la fonction.

Il est permis d'utiliser des vector subscripts pour définir ces sections de tableaux à condition que la procédure ne les modifie pas.

Tableaux de forme explicite comme arguments

La seconde manière d'utiliser des tableaux comme arguments d'une procédure consiste à transmettre leurs dimensions via les arguments de la procédure.

```
FUNCTION PRODUIT_SCALAIRE ( vec1 , vec2 , n )  
  REAL :: PRODUIT_SCALAIRE  
  INTEGER,INTENT(IN) :: n  
  REAL,DIMENSION(1:n),INTENT(IN) :: vec1, vec2
```

```
  REAL,DIMENSION(1:2*n) :: temp
```

Ce n peut être utilisé pour
définir des variables locales.

```
  PRODUIT_SCALAIRE = SUM ( vec1 * vec2 )  
END FUNCTION PRODUIT_SCALAIRE
```

Tableaux de taille implicite comme arguments

Il est possible de laisser libre la dernière dimension d'un tableau passé comme argument d'une procédure. On remplace alors cette dimension par le signe *.

Il n'est pas possible de récupérer cette dimension laissée libre de sorte qu'il est toujours nécessaire de la passer via un argument de la procédure.

```
FUNCTION PRODUIT_SCALAIRE ( vec1 , vec2 , n )
```

```
REAL :: PRODUIT_SCALAIRE
```

```
INTEGER,INTENT(IN) :: n
```

```
REAL,DIMENSION(*),INTENT(IN) :: vec1, vec2
```

```
REAL,DIMENSION(1:2*n) :: temp
```

```
PRODUIT_SCALAIRE = SUM ( vec1(1:n) * vec2(1:n) )
```

```
END FUNCTION PRODUIT_SCALAIRE
```

Il est obligatoire ici de donner la
taille des vecteurs explicitement.

Cette manière de faire ne présente pas beaucoup d'intérêt.

Tableaux de forme implicite comme arguments

La manière la plus souple d'utiliser des tableaux comme arguments d'une procédure consiste à laisser leurs dimensions libres.
On remplace ces dimensions par le signe `::`.

```
FUNCTION PRODUIT_SCALAIRE ( vec1 , vec2 )  
REAL :: PRODUIT_SCALAIRE  
REAL,DIMENSION(:),INTENT(IN) :: vec1, vec2
```

```
REAL,DIMENSION(1:2*SIZE(vec1)) :: temp  
INTEGER :: n  
n=SIZE(vec1)
```

Le n peut être récupéré
de cette manière.

```
PRODUIT_SCALAIRE = SUM ( vec1 * vec2 )  
END FUNCTION PRODUIT_SCALAIRE
```

L'étendue des tableaux est transmise par la procédure qui appelle cette fonction.



Workspace 'Exemple1': 1 proje
+ Exemple1 files

FileView

```
PROGRAM Main
  real,dimension(1:10) :: a, b

  INTERFACE
    FUNCTION PRODUIT_SCALAIRE ( vec1 , vec2 )
      REAL :: PRODUIT_SCALAIRE
      REAL,DIMENSION(:),INTENT(IN) :: vec1, vec2
    END FUNCTION PRODUIT_SCALAIRE
  END INTERFACE

  a = 1. ; b = 1.

  print *, PRODUIT_SCALAIRE ( a , b )

END PROGRAM Main

FUNCTION PRODUIT_SCALAIRE ( vec1 , vec2 )
  REAL :: PRODUIT_SCALAIRE
  REAL,DIMENSION(:),INTENT(IN) :: vec1, vec2

  REAL,DIMENSION(1:2*SIZE(vec1)) :: temp
  INTEGER :: n
  n = SIZE(vec1)

  PRODUIT_SCALAIRE = SUM ( vec1 * vec2 )
END FUNCTION PRODUIT_SCALAIRE
```

"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre6\Debug\Exemple...
10.00000
Press any key to continue

Linking...
Exemple1.exe - 0 error(s), 0 warning(s)

Build Debug Find in Files 1 Find in Files 2

A propos des tableaux allocatable

On peut utiliser des tableaux construits de manière dynamique comme actual arguments. Ils doivent cependant être alloués avant l'appel à la procédure et seront donc traités de la même manière qu'un tableau construit de manière statique.

Les dummy arguments d'une procédure ne peuvent avoir l'attribut allocatable.

Les chaînes de caractères comme arguments

La longueur d'une chaîne de caractères passée comme argument d'une procédure peut être laissée libre. Il suffit de remplacer cette longueur par le signe *.

```
FUNCTION MiseEnForme ( chaine )  
  CHARACTER(LEN=*),INTENT(IN) :: chaine  
  CHARACTER(LEN=LEN(chaine)+4) :: MiseEnForme
```

```
  CHARACTER(LEN=LEN(chaine)) :: temp
```

```
  MiseEnForme = trim(adjustl(chaine))// «.dat»
```

```
END FUNCTION MiseEnForme
```

La longueur de la chaîne de caractères peut être récupérée de cette manière.

La longueur de la chaîne de caractères est transmise par la procédure qui appelle cette fonction.

Exemple3 - Compaq Visual Fortran - [Exemple3.f90]

File Edit View Insert Project Build Tools Window Help

Workspace 'Exemple3': 1 proje
Exemple3 files

```
PROGRAM Main
  INTERFACE
    FUNCTION MiseEnForme ( chaine )
      CHARACTER(LEN=*), INTENT(IN) :: chaine
      CHARACTER(LEN=LEN(chaine)+4) :: MiseEnForme
    END FUNCTION MiseEnForme
  END INTERFACE

  print *, MiseEnForme("input")
  print *, MiseEnForme("output")

END PROGRAM Main

FUNCTION MiseEnForme ( chaine )
  CHARACTER(LEN=*), INTENT(IN) :: chaine
  CHARACTER(LEN=LEN(chaine)+4) :: MiseEnForme

  CHARACTER(LEN=LEN(chaine)) :: temp

  MiseEnForme = trim(adjustl(chaine))//".dat"
END FUNCTION MiseEnForme
```

"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre6\Debug\Exemple..."

```
input.dat
output.dat
Press any key to continue_
```

FileView

Linking...

Exemple3.exe - 0 error(s), 0 warning(s)

Build Debug Find in Files 1 Find in Files 2

Les procédures (sous-routines/fonctions) comme arguments

Si un des arguments est une procédure (sous-routine/fonction), il faut reprendre l'interface de cette procédure.

```
FUNCTION Integration ( f , a , b , n )  
REAL :: Integration  
REAL,INTENT(IN) :: a, b  
INTEGER,INTENT(IN) :: n  
INTERFACE  
FUNCTION f ( x )  
REAL :: f  
REAL,INTENT(IN) :: x  
END FUNCTION f  
END INTERFACE  
REAL :: S, h  
INTEGER :: i  
h = (b-a)/n ; S = 0.  
DO i=1,n-1  
  S = S + f (a+i*h)  
END DO  
Integration = h * ( f (a) / 2. + S + f (b) / 2. )  
END FUNCTION Integration
```

L'argument f est une fonction.

Il faut mettre l'interface de la fonction f.

L'interface de la fonction Integration doit reprendre tout ceci.

```
FUNCTION f (x)  
REAL :: f  
REAL,INTENT(IN) :: x  
f = x * sin(x)  
END FUNCTION f
```

Les procédures (sous-routines/fonctions) comme arguments

La procédure qui utilise Integration devra donc spécifier l'interface de la fonction Integration:

```
INTERFACE  
FUNCTION Integration ( f , a , b , n )  
  REAL :: Integration  
  REAL,INTENT(IN) :: a, b  
  INTEGER,INTENT(IN) :: n  
INTERFACE  
  FUNCTION f ( x )  
    REAL :: f  
    REAL,INTENT(IN) :: x  
  END FUNCTION f  
END INTERFACE  
END FUNCTION Integration  
END INTERFACE
```

de même que l'interface de la fonction f:

```
INTERFACE  
FUNCTION f ( x )  
  REAL :: f  
  REAL,INTENT(IN) :: x  
END FUNCTION f  
END INTERFACE
```

Exemple4 - Compaq Visual Fortran - [Exemple4.f90]

File Edit View Insert Project Build Tools Window Help

num1

Workspace 'Exemple4': 1 proje
Exemple4 files
Exemple4.f90

```

PROGRAM Main
INTERFACE
FUNCTION Integration ( f , a , b , n )
REAL :: Integration
REAL, INTENT(IN) :: a, b
INTEGER, INTENT(IN) :: n
INTERFACE
FUNCTION f ( x )
REAL :: f
REAL, INTENT(IN) :: x
END FUNCTION f
END INTERFACE
END FUNCTION Integration
FUNCTION f ( x )
REAL :: f
REAL, INTENT(IN) :: x
END FUNCTION f
FUNCTION g ( x )
REAL :: g
REAL, INTENT(IN) :: x
END FUNCTION g
END INTERFACE
INTRINSIC SIN
print *, Integration ( f , 0., 3.141592 , 1000 )
print *, Integration ( g , 0., 3.141592 , 1000 )
print *, Integration ( SIN , 0., 3.141592 , 1000 )
END PROGRAM Main
-----
FUNCTION Integration ( f , a , b , n )
REAL :: Integration
REAL, INTENT(IN) :: a, b
INTEGER, INTENT(IN) :: n
INTERFACE
FUNCTION f ( x )
REAL :: f
REAL, INTENT(IN) :: x
END FUNCTION f
END INTERFACE
REAL :: S, h
INTEGER :: i
h = (b-a)/n ; S = 0.
DO i=1,n-1
S = S + f (a+i*h)
END DO
Integration = h * ( f (a) / 2. + S + f (b) / 2. )
END FUNCTION Integration
-----
FUNCTION f(x)
REAL :: f
REAL, INTENT(IN) :: x
f = x * sin(x)
END FUNCTION f
-----
FUNCTION g(x)
REAL :: g
REAL, INTENT(IN) :: x
g = sin(x)
END FUNCTION g

```

INTRINSIC indique au compilateur que SIN est la fonction intrinsèque connue du Fortran.

f, g et SIN sont les « actual arguments » utilisés par le programme qui appelle Integration.

Ce f est le « dummy argument » de la fonction Integration.

```

"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\1-Programmes du cours\Ch...
3.141590
1.999999
1.999999
Press any key to continue_

```

FileView

Configuration: Exemple4 - Win32 Debug

Exemple4.exe - 0 error(s), 0 warning(s)

Build Debug Find in Files 1 Find in Files 2

Ln 29, Col 29 REC COL OVR READ

Les fonctions à valeur de tableau

Le résultat d'une fonction peut être un tableau. Les dimensions de ce tableau peuvent être des constantes, des valeurs transmises explicitement via les arguments de la fonction, ou des valeurs transmises implicitement par la procédure qui fait appel à cette fonction.

```
FUNCTION Transform ( vecteur )  
  REAL,DIMENSION(:),INTENT(IN) :: vecteur  
  REAL,DIMENSION(1:SIZE(VECTEUR)) :: Transform  
  INTEGER :: i, n  
  n = SIZE(VECTEUR)  
  DO i = 1, n  
    Transform(i) = i * vecteur(i)  
  ENDDO  
END FUNCTION Transform
```

La taille du vecteur est récupérée de cette façon. Elle permet alors de définir celle du tableau retourné par la fonction.

Exemple4 - Compaq Visual Fortran - [Exemple4.f90]

File Edit View Insert Project Build Tools Window Help

Workspace 'Exemple4': 1 proje
Exemple4 files
Exemple4.f90

```
PROGRAM Main
  REAL, DIMENSION(1:10) :: Liste = 1.

  INTERFACE
    FUNCTION Transform ( vecteur )
      REAL, DIMENSION(:), INTENT(IN) :: vecteur
      REAL, DIMENSION(1:SIZE(VECTEUR)) :: Transform
    END FUNCTION Transform
  END INTERFACE

  print *, Transform ( Liste )

END PROGRAM Main

FUNCTION Transform ( vecteur )
  REAL, DIMENSION(:), INTENT(IN) :: vecteur
  REAL, DIMENSION(1:SIZE(VECTEUR)) :: Transform
  INTEGER :: i, n
  n = SIZE(VECTEUR)
  DO i = 1, n
    Transform(i) = i * vecteur(i)
  ENDDO
END FUNCTION Transform
```

"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre6\Debug\Exemple..."

1.000000	2.000000	3.000000	4.000000	5.000000
6.000000	7.000000	8.000000	9.000000	10.00000

Press any key to continue_

FileView

Linking...

Exemple4.exe - 0 error(s), 0 warning(s)

Build Debug Find in Files 1 Find in Files 2

Ln 3, Col 36 REC COL OVR READ

Les arguments optionnels

Dans une procédure, les arguments optionnels sont déclarés avec l'attribut OPTIONAL. La fonction intrinsèque PRESENT à valeur logique permet de savoir si l'argument optionnel est présent dans l'appel à la procédure.

```
FUNCTION y (x, n)
```

```
  REAL :: y                                ! spécification de la fonction
```

```
  REAL,INTENT(IN) :: x                    ! spécification des dummy
```

```
  INTEGER,INTENT(IN),OPTIONAL :: n      ! arguments
```

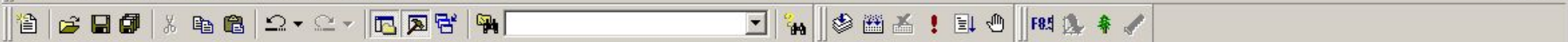
```
  INTEGER :: n$                          ! Spécification des variables locales
```

```
  n$ = 1                                  ! valeur par défaut de la variable locale n$
```

```
  if (PRESENT(n)) n$ = n ! modifier si l'argument optionnel n est spécifié
```

```
  y = x**n$                               ! définir la valeur de la fonction
```

```
END FUNCTION y
```



Workspace 'Exemple4': 1 proje

- Exemple4 files
 - Exemple4.f90

FileView

```
PROGRAM Main
  REAL :: x

  INTERFACE
    FUNCTION y (x, n)
      REAL :: y
      REAL, INTENT(IN) :: x
      INTEGER, INTENT(IN), OPTIONAL :: n
    END FUNCTION y
  END INTERFACE

  x = 2.
  print *, y(x)
  print *, y(x,2)
  print *, y(x,3)
  print *, y(x,4)
  print *, y(x,5)

END PROGRAM Main

FUNCTION y (x, n)
  REAL :: y
  REAL, INTENT(IN) :: x
  INTEGER, INTENT(IN), OPTIONAL :: n
  INTEGER :: n$

  n$ = 1
  if (present(n)) n$ = n

  y = x**n$
END FUNCTION y
```

```
"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre5\Debug\Exemple...
2.000000
4.000000
8.000000
16.00000
32.00000
Press any key to continue
```

-----Configuration: Exemple4 - Win32 Deb
 Exemple4.exe - 0 error(s), 0 warning(s)

Les procédures récursives

Une procédure récursive est une procédure qui peut s'appeler elle-même. Il suffit de la déclarer avec le mot clé RECURSIVE.

```
RECURSIVE FUNCTION Factorielle ( n )  
  INTEGER,INTENT(IN) :: n  
  INTEGER :: Factorielle  
  IF (n<=1) THEN  
    Factorielle = 1  
  ELSE  
    Factorielle = n * Factorielle ( n-1 )  
  ENDIF  
END FUNCTION Factorielle
```

L'utilisation de fonctions récursives est peu efficace.
Il vaut mieux utiliser des boucles.

Exemple5 - Compaq Visual Fortran - [Exemple5.f90]

File Edit View Insert Project Build Tools Window Help

Workspace 'Exemple5': 1 proje
Exemple5 files
Exemple5.f90

```
PROGRAM Main

INTERFACE
  RECURSIVE FUNCTION Factorielle ( n )
    INTEGER, INTENT(IN) :: n
    INTEGER :: Factorielle
  END FUNCTION Factorielle
END INTERFACE

print *, Factorielle ( 5 )

END PROGRAM Main

RECURSIVE FUNCTION Factorielle ( n )
  INTEGER, INTENT(IN) :: n
  INTEGER :: Factorielle
  IF (n<=1) THEN
    Factorielle = 1
  ELSE
    Factorielle = n * Factorielle ( n-1 )
  ENDIF
END FUNCTION Factorielle
```

"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre6\Debug\Exemple...
120
Press any key to continue

FileView

-----Configuration: Exemple5 - Win32 Debug-----
Exemple5.exe - 0 error(s), 0 warning(s)

Build Debug Find in Files 1 Find in Files 2



Les modules

Chapitre 7

Notions générales

Un module est une unité de programme indépendante qui peut contenir les informations ou éléments suivants:

- la précision utilisée (simple ou double précision)
- des instructions de spécification (pour la définition de constantes ou de variables partagées)
- les interfaces de procédures
- des procédures de module

Un module a donc la structure suivante:

```
MODULE <nom>
...           ! instructions de spécification
CONTAINS
...           ! procédures de module
END MODULE <nom>
```

On fait référence à un module avec l'instruction USE <nom>.

Module kinds

On peut définir le type de précision utilisé (simple ou double précision) dans un module kinds.

```
MODULE kinds  
  integer,parameter :: r = 8      ! double précision  
END MODULE kinds
```

On utilisera ensuite ce module en faisant USE kinds.

```
PROGRAM Main  
  use kinds ←  
  real(kind=r) :: x, y  
  ...  
END PROGRAM Main
```

Il faut mettre les use avant les autres instructions de spécification.

Il suffit de modifier la valeur du r dans le module kinds pour changer la précision de tout le programme.

Module de constantes

Un module permet aussi de définir des constantes.

```
MODULE const_fond
```

```
USE kinds
```

```
REAL(kind=r), PARAMETER :: masse = 9.10953E-31_r, &  
                                e      = 1.602189E-19_r, &  
                                hbar   = 1.054589E-34_r, &  
                                pi     = 3.141592654_r
```

```
END MODULE const_fond
```

Ces parametres doivent avoir l'attribut **PARAMETER**.

Le module const_fond se sert du contenu du module kinds.

Module de constantes

On peut alors utiliser tout le module en faisant

```
USE const_fond
```

Pour utiliser une partie seulement des constantes, on peut faire

```
USE const_fond, only: masse, e
```

Pour utiliser une autre nom que celui donné dans le module, on peut faire

```
USE const_fond, only: m => masse
```

Le « masse » du module const_fond sera alors connu sous le nom « m » dans la procédure qui utilise ce module.



Workspace 'Exemple1': 1 proje
Exemple1 files
Exemple1.f90

```
MODULE kinds
  integer,parameter :: r = 8      ! double précision
END MODULE kinds

MODULE const_fond
  use kinds
  real(kind=r),parameter :: masse = 9.10953E-31_r, &
    e = 1.602189E-19_r, &
    hbar = 1.054589E-34_r, &
    pi = 3.141592654_r
END MODULE const_fond

PROGRAM Main
  use kinds
  use const_fond, only: m => masse

  print *, "Mase de l'electron (kg) : ", m
END PROGRAM Main
```

"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre7\Debug\Exemple...
Mase de l'electron (kg) : 9.109530000000000E-031
Press any key to continue_

Linking...
Exemple1.exe - 0 error(s), 0 warning(s)

Module de données partagées

Un module peut également contenir des données (variables, tableaux, etc), qui seront partagées entre plusieurs procédures.

```
MODULE donnees_partagees  
  USE KINDS  
  REAL(kind=r),SAVE :: rayon, hauteur  
END MODULE donnees_partagees
```

Ces données doivent avoir l'attribut SAVE.

On peut à nouveau utiliser tout le module en faisant

```
USE donnees_partagees
```

Pour accéder à une partie seulement des données, on peut faire

```
USE donnees_partagees, only: rayon
```



Workspace 'Exemple2': 1 proje
 Exemple2 files

Exemple2.f90

```

MODULE kinds
  integer,parameter :: r = 8      ! double précision
END MODULE kinds

!
-----
MODULE const_fond
  use kinds
  real(kind=r),parameter :: masse = 9.10953E-31_r, &
    e = 1.602189E-19_r, &
    hbar = 1.054589E-34_r, &
    pi = 3.141592654_r
END MODULE const_fond

!
-----
MODULE donnees_partagees
  use kinds
  real(kind=r),save :: rayon, hauteur
END MODULE donnees_partagees

!
PROGRAM Main
  use kinds
  use donnees_partagees

  INTERFACE
    FUNCTION Poids ( rho )
      use kinds
      real(kind=r) :: Poids
      real(kind=r),intent(in) :: rho
    END FUNCTION Poids
  END INTERFACE

  real(kind=r) :: rho = 1000._r

  rayon = 2.
  hauteur = 10.

  print *, "Poids du cylindre (kg) = ", Poids ( rho )

END PROGRAM Main

!
-----
FUNCTION Poids ( rho )
  use kinds
  use const_fond, only: pi
  use donnees_partagees

  real(kind=r) :: Poids
  real(kind=r),intent(in) :: rho

  Poids = rho * (pi * rayon**2 * hauteur)
END FUNCTION Poids

```

"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre7\Debug\Exemple...

```

Poids du cylindre (kg) = 125663.706160000
Press any key to continue

```

FileView

-----Configuration: Exemple2 - Win32 Debug-----
 Linking...
 Exemple2.exe - 0 error(s), 0 warning(s)
 Build Debug Find in Files 1 Find in Files 2

Module contenant des interfaces

Un module peut rassembler les interfaces de toutes les procédures rencontrées dans un projet.

```
MODULE list_interface
INTERFACE
FUNCTION Poids ( rho )
  use kinds
  real(kind=r) :: Poids
  real(kind=r),intent(in) :: rho
END FUNCTION Poids
...
END INTERFACE
END MODULE list_interface
```

} ouverture du module

} liste de toutes les interfaces du projet

} fermeture du module

L'avantage est qu'il suffit de construire un seul exemplaire de ces interfaces, même si les procédures en question sont utilisées à plusieurs reprises dans le projet.

Module contenant des interfaces

On peut utiliser tout le module en faisant

```
USE list_interface
```

Il est préférable cependant de faire référence, dans chaque procédure, aux interfaces réellement nécessaires.

```
USE list_interface, only: Poids
```

Il faut par ailleurs éviter qu'une procédure importe sa propre interface.


```
Exemple3 - Compaq Visual Fortran - [Exemple3.f90]
File Edit View Insert Project Build Tools Window Help
Workspace 'Exemple3': 1 proje
  Exemple3 files
    Exemple3.f90

MODULE kinds
  integer,parameter :: r = 8      ! double précision
END MODULE kinds

MODULE const_fond
  use kinds
  real(kind=r),parameter :: masse = 9.10953E-31_r, &
    e = 1.602189E-19_r, &
    hbar = 1.054589E-34_r, &
    pi = 3.141592654_r
END MODULE const_fond

MODULE donnees_partagees
  use kinds
  real(kind=r),save :: rayon, hauteur
END MODULE donnees_partagees

MODULE list_interface
  INTERFACE
    FUNCTION Poids ( rho )
      use kinds
      real(kind=r) :: Poids
      real(kind=r),intent(in) :: rho
    END FUNCTION Poids
  END INTERFACE
END MODULE list_interface

PROGRAM Main
  use kinds
  use donnees_partagees
  use list_interface

  real(kind=r) :: rho = 1000._r

  rayon = 2.
  hauteur = 10.

  print *, "Poids du cylindre (kg) = ", Poids ( rho )

END PROGRAM Main

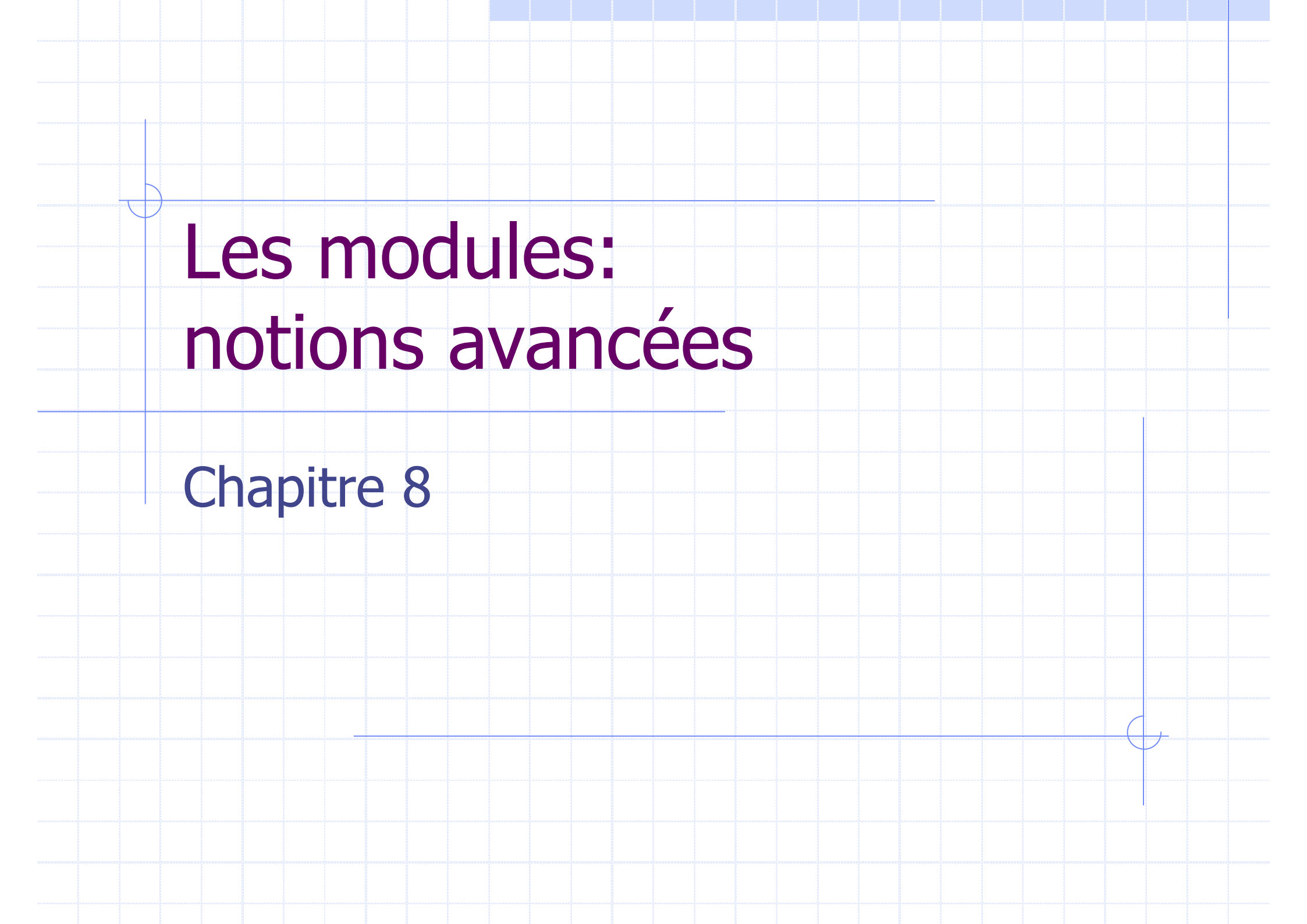
FUNCTION Poids ( rho )
  use kinds
  use const_fond, only: pi
  use donnees_partagees

  real(kind=r) :: Poids
  real(kind=r),intent(in) :: rho

  Poids = rho * (pi * rayon**2 * hauteur)
END FUNCTION Poids
```

```
"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre 7\Debug\Exemple...
Poids du cylindre (kg) = 125663.706160000
Press any key to continue
```

```
-----Configuration: Exemple3 - Win32 Debug-----
Linking...
Exemple3.exe - 0 error(s), 0 warning(s)
Build Debug Find in Files 1 Find in Files 2
```



Les modules: notions avancées

Chapitre 8

Les procédures de module

Un module peut encapsuler des procédures. L'avantage de cette façon de travailler est qu'il n'est plus nécessaire de créer des interfaces.

```
MODULE Module_Poids
```

```
CONTAINS
```

```
FUNCTION Poids ( rho )
```

```
  use kinds
```

```
  use const_fond, only: pi
```

```
  use donnees_partagees
```

```
  real(kind=r) :: Poids
```

```
  real(kind=r),intent(in) :: rho
```

```
  Poids = rho * (pi * rayon**2 * hauteur)
```

```
END FUNCTION Poids
```

```
END MODULE Module_Poids
```

On met ici la procédure dans son entièreté. Il est possible d'en mettre plusieurs.

Les procédures de module

Afin d'utiliser toute les procédures contenues dans ce module, il suffit de faire

```
USE Module_Poids
```

Pour accéder à une procédure particulière de ce module, on peut faire par exemple

```
USE Module_Poids, only: Poids
```

Une procédure de module peut contenir des procédures internes.

Workspace 'Exemple2': 1 proje
Exemple2 files
Exemple2.f90

```
MODULE kinds
  integer, parameter :: r = 8      ! double précision
END MODULE kinds

-----
MODULE const_fond
  use kinds
  real(kind=r), parameter :: masse = 9.10953E-31_r, &
    e = 1.602189E-19_r, &
    hbar = 1.054589E-34_r, &
    pi = 3.141592654_r
END MODULE const_fond

-----
MODULE donnees_partagees
  use kinds
  real(kind=r), save :: rayon, hauteur
END MODULE donnees_partagees

-----
MODULE Module_Poids
CONTAINS
  FUNCTION Poids ( rho )
    use kinds
    use const_fond, only: pi
    use donnees_partagees

    real(kind=r) :: Poids
    real(kind=r), intent(in) :: rho

    Poids = rho * (pi * rayon**2 * hauteur)

  END FUNCTION Poids
END MODULE Module_Poids

-----
PROGRAM Main
  use kinds
  use donnees_partagees
  use Module_Poids

  real(kind=r) :: rho = 1000._r

  rayon = 2.
  hauteur = 10.

  print *, "Poids du cylindre (kg) = ", Poids ( rho )
END PROGRAM Main
```

```
"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre8\Debug\Exemple...
Poids du cylindre (kg) = 125663.706160000
Press any key to continue
```

Configuration: Exemple2 - Win32 Debug

Exemple2.exe - 0 error(s), 0 warning(s)

Build Debug Find in Files 1 Find in Files 2

Les interfaces génériques

Beaucoup d'instructions intrinsèques, comme $\text{SIN}(x)$, ont des noms génériques. Cela signifie qu'on peut les utiliser avec des arguments de type et de rangs différents. Par exemple, x peut être en simple ou en double précision, x peut être une variable simple ou un tableau. Toutes ces versions de la fonction SIN sont regroupées sous un seul nom.

Il est possible de créer des procédures qui associent également, sous un seul nom, différentes versions.

Une première façon de travailler consiste à associer ces différentes versions via une interface générique.

Les interfaces génériques

```
INTERFACE xsin ← nom générique
  FUNCTION xsin_sp(x)
    REAL(kind=4),INTENT(IN) :: x
    REAL(kind=4) :: xsin_sp
  END FUNCTION xsin_sp
  FUNCTION xsin_dp(x)
    REAL(kind=8),INTENT(IN) :: x
    REAL(kind=8) :: xsin_dp
  END FUNCTION xsin_dp
END INTERFACE xsin
```

version en simple précision

version en double précision

Cette interface peut être incluse dans la procédure qui utilise xsin ou dans un module list_interface.

Les interfaces génériques

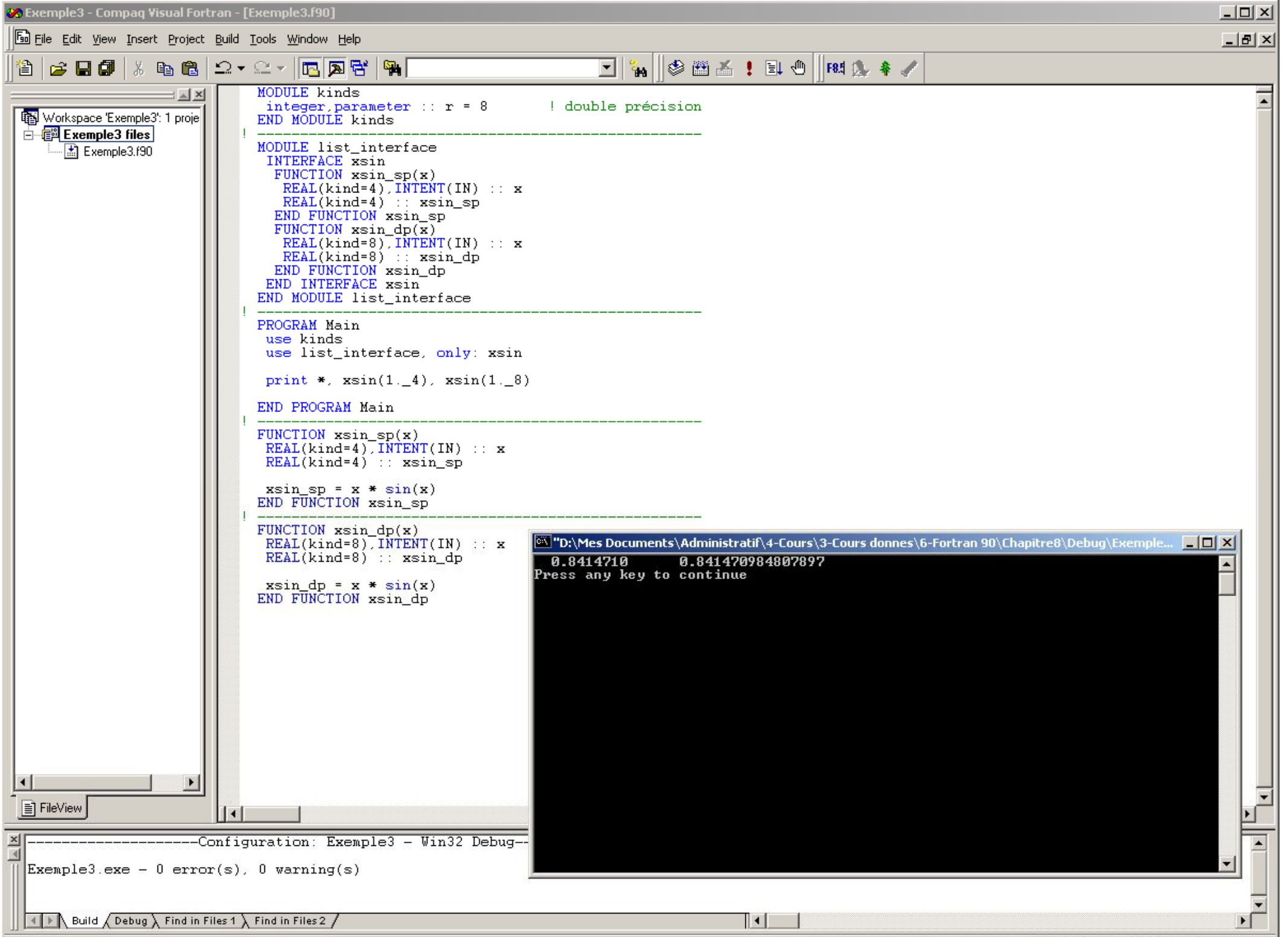
```
MODULE list_interface
INTERFACE xsin ← nom générique
  FUNCTION xsin_sp(x)
    REAL(kind=4),INTENT(IN) :: x
    REAL(kind=4) :: xsin_sp
  END FUNCTION xsin_sp
  FUNCTION xsin_dp(x)
    REAL(kind=8),INTENT(IN) :: x
    REAL(kind=8) :: xsin_dp
  END FUNCTION xsin_dp
END INTERFACE xsin
END MODULE list_interface
```

version en simple précision

version en double précision

On fait référence à cette interface générique par l'instruction

```
USE list_interface, only: xsin
```

Les interfaces génériques avec procédures de module

Une deuxième façon de travailler consiste à placer les différentes versions de la procédure dans un module. Ces procédures deviennent donc des procédures de module.

On associe ensuite ces différentes versions sous un nom commun via une interface générique.

Les interfaces génériques avec procédures de module

```
MODULE Module_xsin
  INTERFACE xsin ← nom générique
    MODULE PROCEDURE xsin_sp, xsin_dp
  END INTERFACE xsin
CONTAINS
  FUNCTION xsin_sp(x)
    REAL(kind=4),INTENT(IN) :: x
    REAL(kind=4) :: xsin_sp
    xsin_sp = x * sin(x)
  END FUNCTION xsin_sp
  FUNCTION xsin_dp(x)
    REAL(kind=8),INTENT(IN) :: x
    REAL(kind=8) :: xsin_dp
    xsin_dp = x * sin(x)
  END FUNCTION xsin_dp
END MODULE Module_xsin
```

version en simple précision

version en double précision

Les interfaces génériques avec procédures de module

On fait ensuite référence à ce module avec l'instruction

```
USE Module_xsin
```

Les noms génériques peuvent coïncider avec le nom de fonctions intrinsèques du Fortran (overloading). Les procédures regroupées sous un nom commun doivent toutes être des sous-routines ou toutes être des fonctions.

Exemple4 - Compaq Visual Fortran - [Exemple4.f90]

File Edit View Insert Project Build Tools Window Help

Workspace 'Exemple4': 1 proje
Exemple4 files
Exemple4.f90

```
MODULE kinds
  integer, parameter :: r = 8      ! double précision
END MODULE kinds

-----
MODULE Module_xsin
  INTERFACE xsin
    MODULE PROCEDURE xsin_sp, xsin_dp
  END INTERFACE xsin
CONTAINS
  FUNCTION xsin_sp(x)
    REAL(kind=4), INTENT(IN) :: x
    REAL(kind=4) :: xsin_sp
    xsin_sp = x * sin(x)
  END FUNCTION xsin_sp
  FUNCTION xsin_dp(x)
    REAL(kind=8), INTENT(IN) :: x
    REAL(kind=8) :: xsin_dp
    xsin_dp = x * sin(x)
  END FUNCTION xsin_dp
END MODULE Module_xsin

-----
PROGRAM Main
  use kinds
  use Module_xsin

  print *, xsin(1._4), xsin(1._8)

END PROGRAM Main
```

cmd "D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre8\Debug\Exemple..."
0.8414710 0.841470984807897
Press any key to continue

FileView

-----Configuration: Exemple4 - Win32 Debug-----
Exemple4.exe - 0 error(s), 0 warning(s)

Build Debug Find in Files 1 Find in Files 2

Définition d'opérateurs

Il est possible d'étendre la définition d'opérateurs existants (+, -, *, /) ou de créer ses propres opérateurs (.cross., .conv., etc). Le nom de ces derniers doit obligatoirement commencer et terminer par un point.

Ces opérateurs sont en réalité associés à des procédures, dont les arguments doivent avoir l'attribut INTENT(IN). On peut construire des opérateurs à un ou deux arguments.

Ces opérateurs sont définis par un module.

Définition d'opérateurs

```
MODULE Module_cross
  INTERFACE OPERATOR (.cross.)
  MODULE PROCEDURE cross
  END INTERFACE
CONTAINS
  FUNCTION cross ( vec1 , vec2 )
    use kinds
    real(kind=r),dimension(1:3),intent(in) :: vec1, vec2
    real(kind=r),dimension(1:3) :: cross
    cross(1) = vec1(2) * vec2(3) - vec1(3) * vec2(2)
    cross(2) = vec1(3) * vec2(1) - vec1(1) * vec2(3)
    cross(3) = vec1(1) * vec2(2) - vec1(2) * vec2(1)
  END FUNCTION cross
END MODULE Module_cross
```

← nom de l'opérateur

← nom de la fonction qui implémente l'opérateur

} définition de la fonction cross qui implémente l'opérateur .cross.

Définition d'opérateurs

Pour utiliser l'opérateur `.cross.`, il suffit alors de faire

```
USE Module_cross
```

Il faut noter que `.cross.` correspond au nom de l'opérateur et `cross` au nom de la fonction qui l'implémente. Le choix de ces noms est indépendant.

De manière similaire à ce qui a été vu précédemment, on peut définir un `cross_sp` et un `cross_dp` pour traiter la simple et la double précision. L'opérateur `.cross.` devient alors un opérateur générique pour ces deux cas de figure.

Définition d'opérateurs

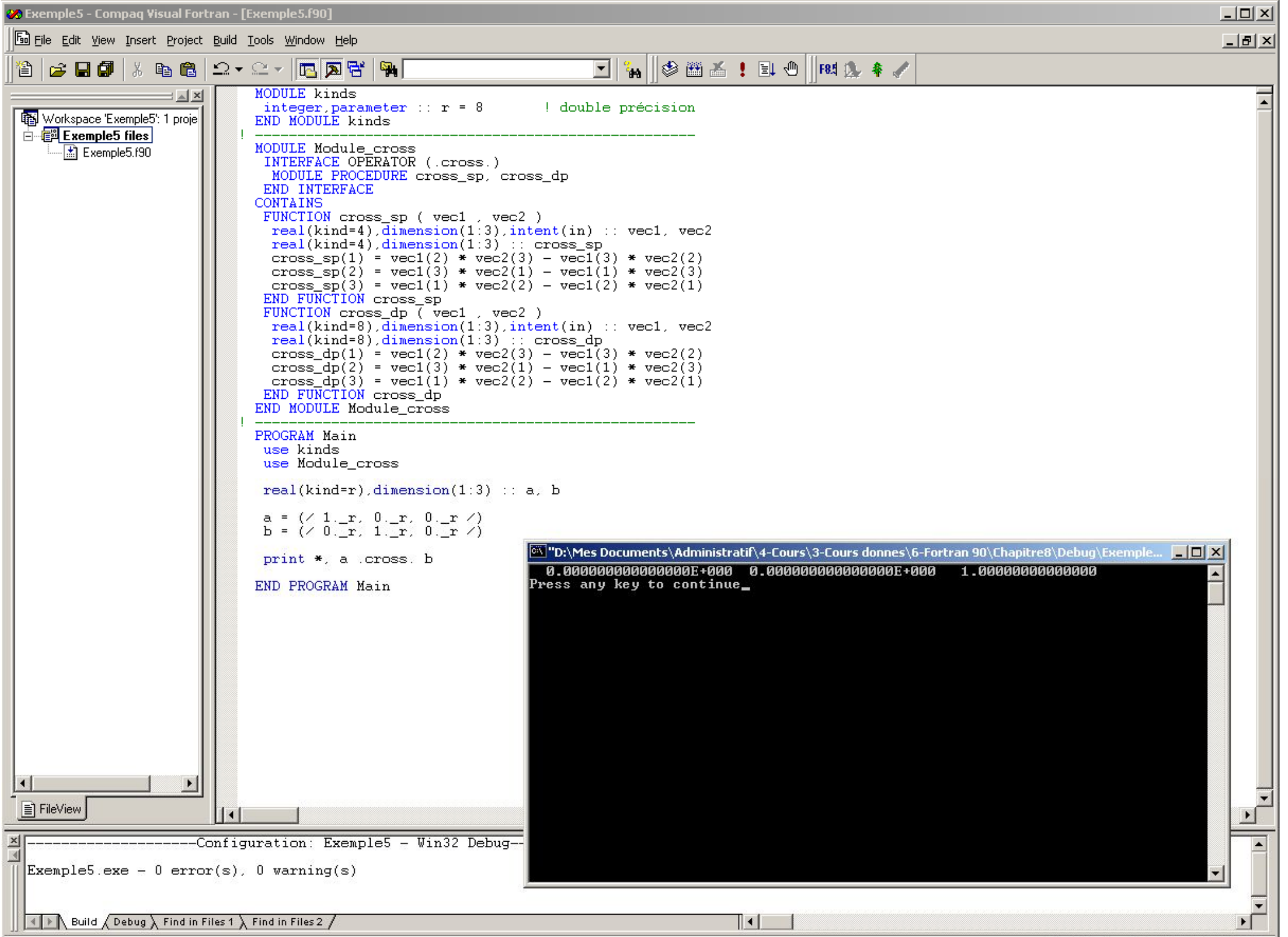
```
MODULE Module_cross
  INTERFACE OPERATOR (.cross.)
    MODULE PROCEDURE cross_sp, cross_dp
  END INTERFACE
CONTAINS
  FUNCTION cross_sp ( vec1 , vec2 )
    real(kind=4),dimension(1:3),intent(in) :: vec1, vec2
    real(kind=4),dimension(1:3) :: cross_sp
    ...
  END FUNCTION cross_sp
  FUNCTION cross_dp ( vec1 , vec2 )
    real(kind=8),dimension(1:3),intent(in) :: vec1, vec2
    real(kind=8),dimension(1:3) :: cross_dp
    ...
  END FUNCTION cross_dp
END MODULE Module_cross
```

← nom de l'opérateur

← nom des fonctions qui implémentent l'opérateur

} implémentation de l'opérateur en simple précision

} implémentation de l'opérateur en double précision



Définition d'opérateurs

On insistera à nouveau sur le fait qu'on peut étendre la portée d'opérateurs intrinsèques du Fortran, mais jamais remplacer des cas de figure déjà implémentés. Par exemple, on peut étendre la portée de l'opérateur + pour traiter des structures composites, mais il n'est pas permis de redéfinir l'action du + entre deux nombres réels.

Il est également possible d'étendre la portée du signe d'affectation (=) ou de créer ses propres opérations d'affectation. On utilise pour cela un bloc INTERFACE ASSIGNMENT (=). Ici aussi, il n'est pas permis de redéfinir des cas déjà implémentés.

Les attributs PRIVATE et PUBLIC

Par défaut, toutes les variables définies dans un module sont accessibles par les procédures qui y font appel. Ces variables ont implicitement l'attribut PUBLIC.

On peut donner à certaines variables l'attribut PRIVATE. Leur valeur n'est alors utilisable que dans ce module. Ces variables ne servent qu'à définir d'autres variables contenues dans ce module.

```
MODULE AtomicUnits
```

```
USE kinds
```

```
REAL(kind=r),PARAMETER,PRIVATE :: e      = 1.602189E-19_r , &  
                                     masse = 9.10953E-31_r , &  
                                     hbar   = 1.054589E-34_r , &  
                                     pi     = 3.141592654_r , &  
                                     Epsilon0 = 8.85418782E-12_r
```

```
REAL(kind=r),PARAMETER :: rBohr = 4._r*pi*Epsilon0*hbar**2 &  
                                / (masse*e**2)
```

```
END MODULE AtomicUnits
```



Workspace 'Exemple1': 1 proje
Exemple1 files
Exemple1.f90

```
MODULE kinds
  integer,parameter :: r = 8      ! double précision
END MODULE kinds

-----
MODULE AtomicUnits
  USE kinds
  REAL(kind=r),PARAMETER,PRIVATE :: e      = 1.602189E-19_r, &
    masse      = 9.10953E-31_r, &
    hbar       = 1.054589E-34_r, &
    pi        = 3.141592654_r, &
    Epsilon0  = 8.85418782E-12_r
  REAL(kind=r),PARAMETER :: rBohr = 4._r*pi*Epsilon0*hbar**2 / (masse*e**2)
END MODULE AtomicUnits

-----
PROGRAM Main
  use AtomicUnits
  print *, rBohr, e
END PROGRAM Main
```

```
"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\Chapitre8\Debug\Exemple...
5.291777880150987E-011  0.0000000E+00
Press any key to continue
```

Les attributs PRIVATE et PUBLIC

Une formulation alternative est la suivante:

```
MODULE AtomicUnits
```

```
USE kinds
```

```
REAL(kind=r),PARAMETER :: e      = 1.602189E-19_r, &  
                           masse = 9.10953E-31_r, &  
                           hbar   = 1.054589E-34_r, &  
                           pi     = 3.141592654_r, &  
                           Epsilon0 = 8.85418782E-12_r
```

```
REAL(kind=r),PARAMETER :: rBohr = 4._r*pi*Epsilon0*hbar**2 &  
                               / (masse*e**2)
```

```
PRIVATE :: e, masse, hbar, pi, Epsilon0
```

```
END MODULE AtomicUnits
```



On mentionne les variables
que l'on veut PRIVATE.

Les attributs PRIVATE et PUBLIC

Ou encore:

```
MODULE AtomicUnits  
USE kinds  
PRIVATE
```

Par défaut, toutes les variables
sont PRIVATE ...

```
REAL(kind=r),PARAMETER :: e      = 1.602189E-19_r, &  
                           masse = 9.10953E-31_r, &  
                           hbar   = 1.054589E-34_r, &  
                           pi      = 3.141592654_r, &  
                           Epsilon0 = 8.85418782E-12_r  
REAL(kind=r),PARAMETER,PUBLIC :: rBohr=4._r*pi*Epsilon0*hbar**2 &  
                               /(masse*e**2)
```

```
END MODULE AtomicUnits
```

... sauf celles déclarées
explicitement PUBLIC.



Les pointeurs

Chapitre 9

Les pointeurs: notions générales

Un pointeur (POINTER) indique l'adresse mémoire d'un objet cible (TARGET). L'objet cible peut être une variable, un tableau, une structure composite, etc.

On peut travailler avec le pointeur comme on travaillerait avec les objets cibles. En manipulant une seule entité (le pointeur), on peut donc agir sur différents objets dans le programme (les cibles).

Le pointeur et les cibles doivent être déclarés avec le même type, le même kind et le même rang. Les pointeurs reçoivent l'attribut POINTER et les objets l'attribut TARGET.

Les pointeurs: déclaration et association

Déclaration:

```
REAL,DIMENSION(:),POINTER :: liste
```

```
REAL,DIMENSION(:),ALLOCATABLE,TARGET :: liste1, liste2
```

Pour pointer vers liste1 ou liste2, on fait

```
liste => liste1
```

ou

```
liste => liste2
```

Pour savoir si un pointeur est associé, on peut utiliser la fonction logique ASSOCIATED:

```
ASSOCIATED (liste)
```

donne une valeur vraie si le pointeur est associé et une valeur fausse sinon.

Les pointeurs: annulation

Pour annuler l'association entre un pointeur et sa cible, on peut faire

NULLIFY (liste)



Workspace 'Exemple1': 1 proje

- Exemple1 files
 - Exemple1.f90

```

Program Main
real,dimension(:),pointer :: liste
real,dimension(:),allocatable,target :: liste1, liste2

allocate (liste1(1:100),liste2(1:200)) ! allouer les tableaux

liste => liste1 ! association
print *, size(liste)

liste => liste2 ! association
print *, size(liste)

if (associated(liste)) then ! tester l'association
  print *, "Le pointeur est associe."
else
  print *, "Le pointeur est dissocie."
endif

nullify (liste) ! annuler l'association

deallocate (liste1, liste2) ! deallouer les tableaux

if (associated(liste)) then ! tester l'association
  print *, "Le pointeur est associe."
else
  print *, "Le pointeur est dissocie."
endif

end Program Main
  
```

```

"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\1-Programmes du cours\Ch...
100
200
Le pointeur est associe.
Le pointeur est dissocie.
Press any key to continue
  
```

-----Configuration: Exemple1 - Win32 Debug-----

Exemple1.exe - 0 error(s), 0 warning(s)

Build Debug Find in Files 1 Find in Files 2

Les pointeurs: considérations finales

Le Fortran permet d'utiliser le pointeur comme un tableau allouable. On lui attribue alors une existence propre, indépendante des cibles qu'il pourrait pointer.

```
ALLOCATE (liste(1:300))  
print *, SIZE(liste)  
DEALLOCATE(liste)
```

Pour des questions d'efficacité, il peut être intéressant d'utiliser un pointeur comme argument d'une procédure. On transmet ainsi à cette procédure l'adresse mémoire de l'objet à manipuler plutôt que l'objet lui-même.



La librairie DFLIB

Chapitre 10

Implémenter la librairie DFLIB

La librairie DFLIB permet de réaliser des graphiques en Fortran.

Il est nécessaire pour cela de créer un projet du type « Fortran Standard Graphics or QuickWin Application » et de choisir « QuickWin (multiple windows) ».

On fait référence à la librairie avec l'instruction

```
USE DFLIB
```

Celle-ci doit être placée avant les premières instructions de spécification.

Les systèmes de coordonnées: les coordonnées du texte

Les coordonnées du texte :

lignes : de 1 à 25 (selon l'écran)

colonnes : de 1 à 80 (selon l'écran)

position : (ligne , colonne)

Ligne 1

Colonne 1

Ligne 1

Colonne 80

Ligne 25

Colonne 1

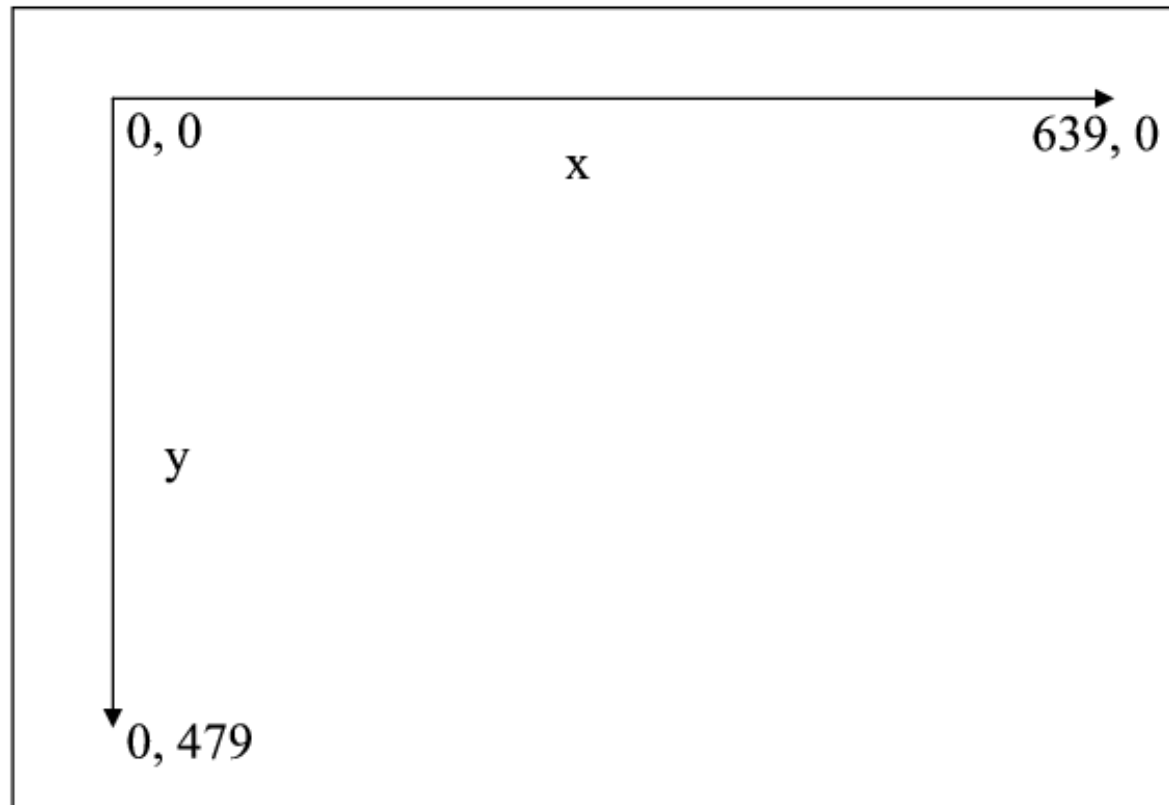
Les systèmes de coordonnées: les coordonnées physiques

Les coordonnées physiques :

x : de 0 à 639 pixels (selon l'écran)

y : de 0 à 479 pixels (selon l'écran)

position : (x, y) où x et y sont des INTEGER(kind=2)
et $(0, 0)$ correspond au coin supérieur gauche



Changer l'origine du système de coordonnées physiques

On peut changer l'origine du système de coordonnées avec la sous-routine SETVIEWORG, dont les deux premiers arguments sont des INTEGER(kind=2).

```
CALL SETVIEWORG (50_2, 100_2, xy )
```

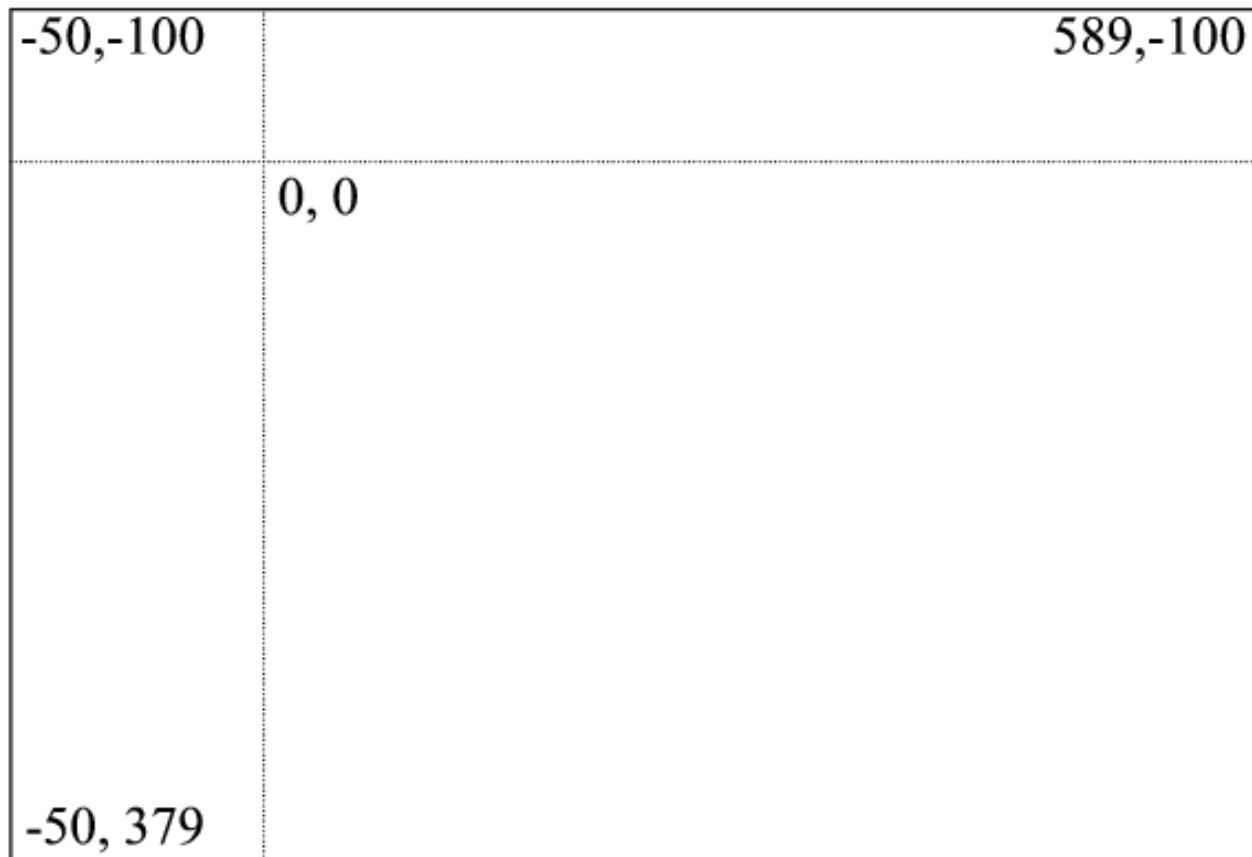
xy contient en sortie les coordonnées physiques de l'ancienne origine. Dans le cas présent, ce serait (0,0). xy doit être déclaré comme

```
TYPE ( xycoord ) xy
```

Le type xycoord est défini dans la librairie DFLIB.

Changer l'origine du système de coordonnées physiques

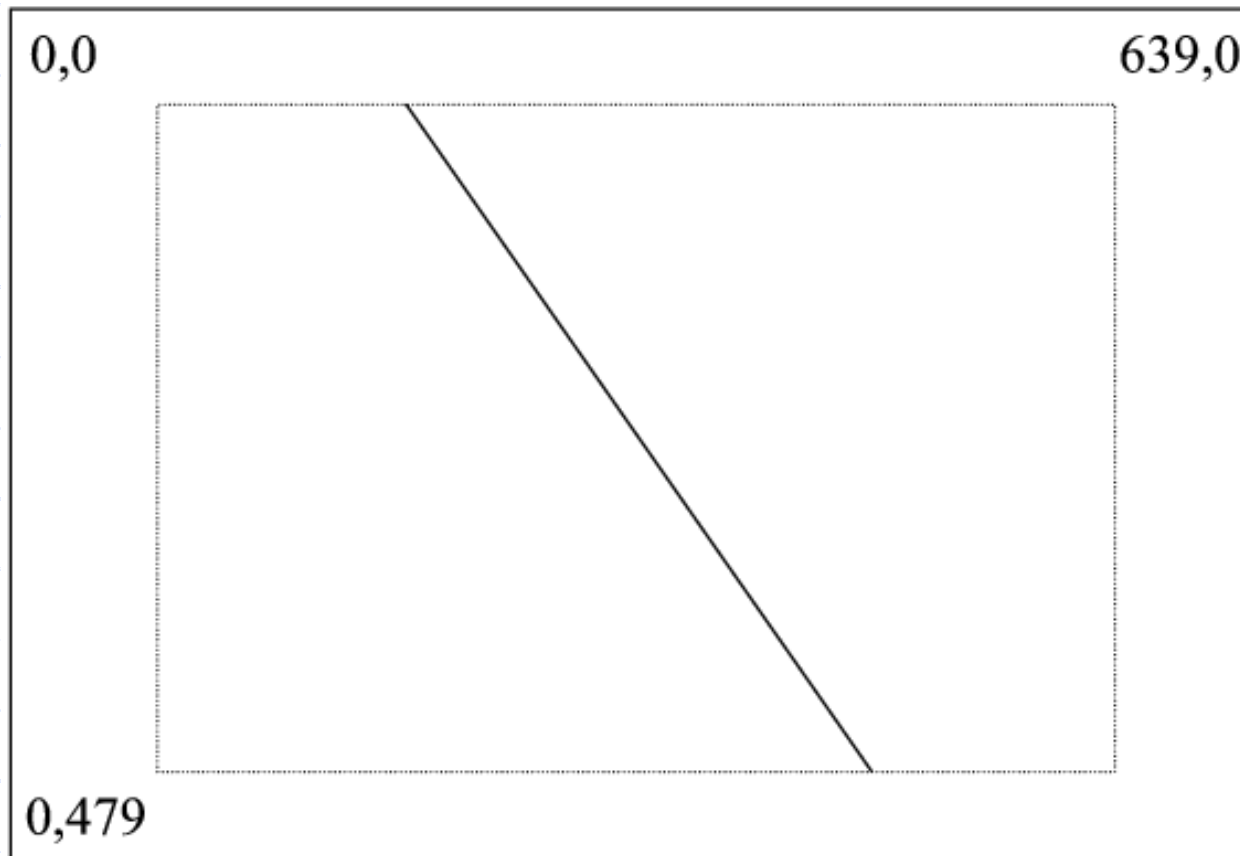
L'action du `CALL SETVIEWORG (50_2, 100_2, xy)` est donc de mettre l'origine en (50 , 100) par rapport au système de coordonnées physiques.



Définir une « clipping region »

On peut définir une « clipping region », à l'extérieur de laquelle il sera interdit de dessiner.

```
CALL SETCLIPRGN ( 30_2 , 30_2 , 609_2 , 449_2 )
```



La commande SETVIEWPORT

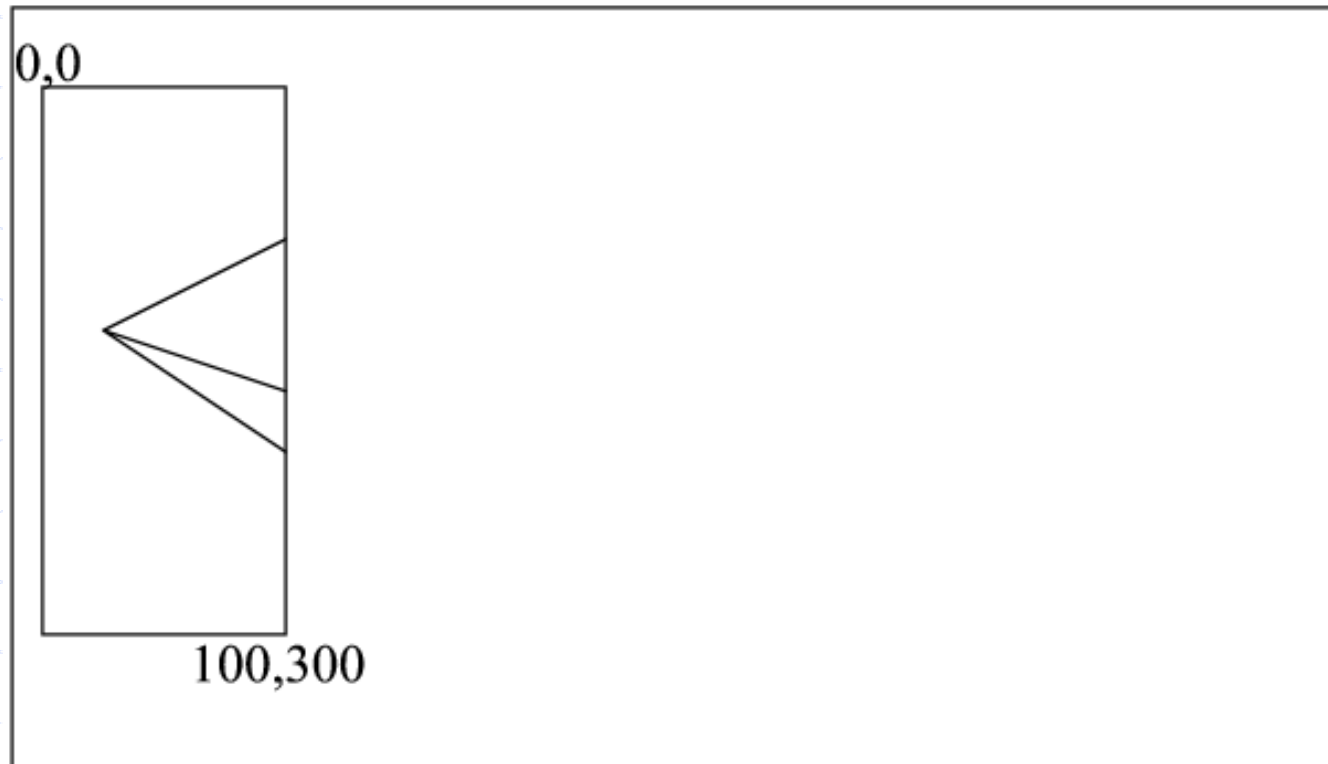
La commande SETVIEWPORT combine les deux instructions précédentes en définissant une fenêtre à l'extérieur de laquelle on ne pourra plus dessiner et en plaçant la nouvelle origine au coin supérieur gauche de cette fenêtre.

```
CALL SETVIEWPORT ( 0_2 , 0_2 , 100_2 , 300_2 )
```

Les quatre arguments de SETVIEWPORT doivent être des INTEGER(kind=2).

La commande SETVIEWPORT

L'effet du `CALL SETVIEWPORT (0_2 , 0_2 , 100_2 , 300_2)` est donc de restreindre la fenêtre de représentation au rectangle sous-tendu par les points (0,0) et (100,300) par rapport au système physique.



L'origine est placée au coin supérieur gauche de cette fenêtre.

Les systèmes de coordonnées: les coordonnées réelles

La commande SETWINDOW permet de changer l'échelle de la viewport en associant des « valeurs réelles » aux limites de la fenêtre de représentation.

On utilise SETWINDOW comme

```
dummy2 = SETWINDOW (.true.,xmin,ymin,xmax,ymax)
```

où dummy2 est un INTEGER(kind=2) et xmin, ymin, xmax et ymax sont des REAL(kind=8).

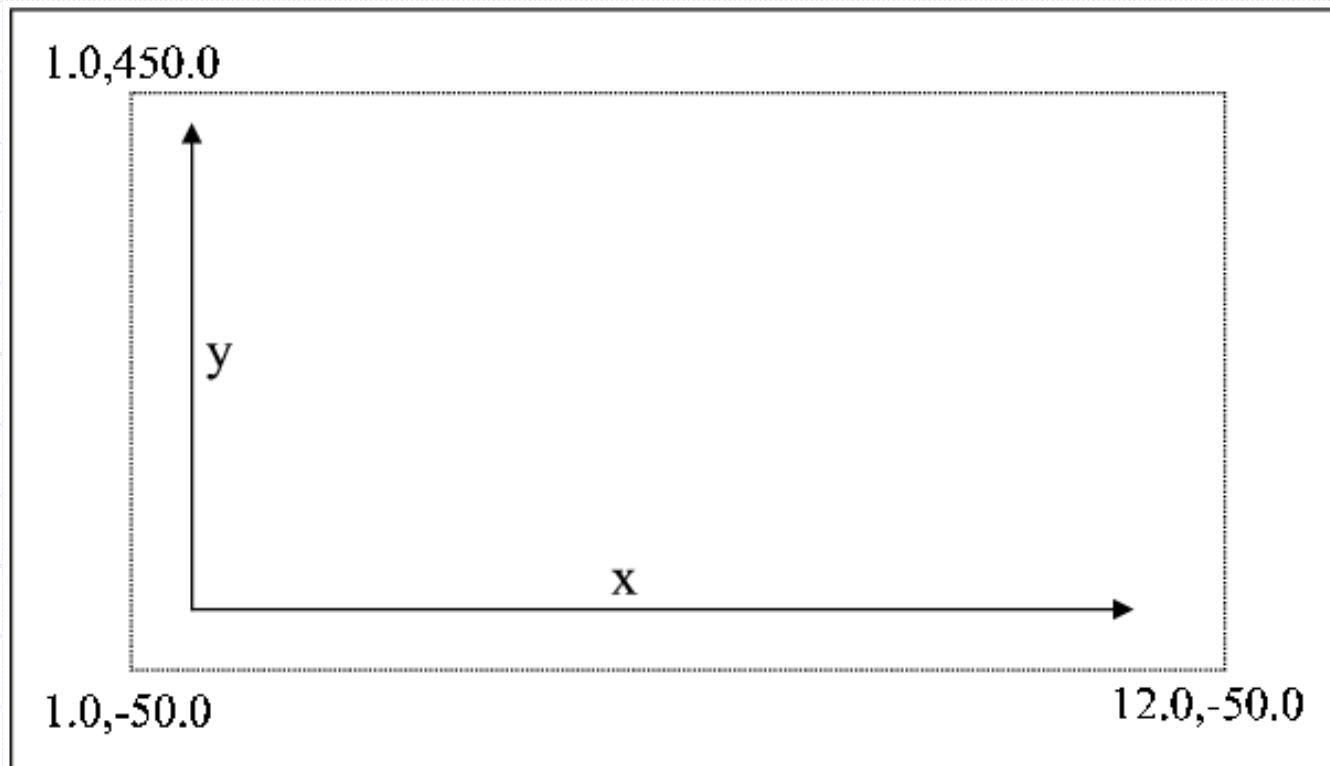
(xmin,ymin) définit la valeur du coin inférieur gauche de la fenêtre de représentation dans le « système de coordonnées réelles ».

(xmax,ymax) redéfinit la valeur du coin supérieur droit de la fenêtre de représentation.

Les « coordonnées réelles » sont toujours des REAL(kind=8).

Les systèmes de coordonnées

L'effet de `dummy2 = SETWINDOW (.true.,1._8,-50._8,12._8,450._8)` serait donc d'appliquer à la viewport un système de coordonnées réelles défini comme dans la figure suivante.



Effacer l'écran

Pour effacer tout l'écran, on peut faire

```
call clearscreen ($Gclearscreen)
```

Pour effacer la viewport seulement, on peut faire

```
call clearscreen ($Gviewport)
```

Initialisation d'un programme graphique

Les premières lignes d'un programme graphique pourraient donc ressembler à ceci:

Program Main

```
use DFLIB
```

```
integer(kind=2) :: dummy2
```

```
integer(kind=2) :: resx = 640, resy = 480
```

```
real(kind=8) :: xmin, xmax, ymin, ymax
```

```
xmin = 0._8 ; xmax = 10._8 ; ymin = -1._8 ; ymax = 1._8
```

```
call SETVIEWPORT ( 4_2 , 4_2 , resx-5_2 , resy-5_2 )
```

```
dummy2 = SETWINDOW (.true. , xmin , ymin , xmax , ymax )
```

```
call clearscreen($Gclearscreen)
```

```
end Program Main
```

effacer l'écran



Changer la couleur

Pour choisir la couleur:

```
dummy2 = setcolor (color)
```

où dummy2 et color sont des INTEGER(kind=2). color correspond à des couleurs prédéfinies et peut prendre des valeurs comprises entre 0 et 16.

Une manière plus fine de préciser la couleur est la suivante:

```
dummy2 = setcolorrgb (color)
```

où color est cette fois un INTEGER(kind=4) qui prend la valeur #B|G|R où B, G et R représentent les niveaux de bleu, vert et rouge en hexadécimal (valeurs comprises entre 00 et FF).

Changer la couleur

Une manière plus fine de préciser la couleur est la suivante:

```
dummy2 = setcolorrgb (color)
```

où color est cette fois un INTEGER(kind=4) qui prend la valeur #B|G|R où B, G et R représentent les niveaux de bleu, vert et rouge en hexadécimal (valeurs comprises entre 00 et FF).

Concrètement :

```
INTEGER(kind=4) :: blue, green, red, color
```

```
color = red + 256 * green + 256 * 256 * blue
```

avec blue, green, red \in [0,255].

Changer la couleur du fond d'écran

Pour choisir la couleur du fond d'écran:

```
dummy2 = setbkcolor (color)
```

où dummy2 et color sont des INTEGER(kind=2). color correspond à des couleurs prédéfinies et peut prendre des valeurs comprises entre 0 et 16.

Une manière plus fine de préciser la couleur du fond d'écran est la suivante:

```
dummy2 = setbkcolorrgb (color)
```

où color est cette fois un INTEGER(kind=4) qui prend la valeur #B|G|R où B, G et R représentent les niveaux de bleu, vert et rouge en hexadécimal (valeurs comprises entre 00 et FF).

Il faut effacer l'écran pour voir ces instructions prendre effet.

Relier deux points en coordonnées physiques

Pour relier les points (x_1, y_1) et (x_2, y_2) en coordonnées physiques, les instructions suivantes sont nécessaires:

```
integer(kind=2) :: dummy2, x1, y1, x2, y2  
type(xycoord) :: xy
```

```
call moveto (x1, y1, xy)  
dummy2 = lineto (x2, y2)
```

L'instruction `moveto` sert à définir le point de départ. L'instruction `lineto` relie alors (x_1, y_1) au point suivant. On peut continuer à utiliser `lineto` tant qu'il y a des points à relier. `xy` conserve à chaque fois le dernier point atteint.

Relier deux points en coordonnées réelles

Pour relier les points $(wx1,wy1)$ et $(wx2,wy2)$ en coordonnées réelles, les instructions suivantes sont nécessaires:

```
integer(kind=2) :: dummy2  
real(kind=8) :: wx1, wy1, wx2, wy2  
type(wxycoord) :: wxy
```

```
call moveto_w (wx1, wy1, wxy)  
dummy2 = lineto_w (wx2,wy2)
```

L'instruction `moveto_w` sert à définir le point de départ. L'instruction `lineto_w` relie alors $(wx1,wy1)$ au point suivant. On peut continuer à utiliser `lineto_w` tant qu'il y a des points à relier. `wxy` conserve à chaque fois le dernier point atteint.

Dessiner un rectangle

En coordonnées physiques :

```
dummy2 = rectangle (control, x1, y1, x2, y2)
```

En coordonnées réelles:

```
dummy2 = rectangle_w (control, wx1, wy1, wx2, wy2)
```

Pour remplir le rectangle, on prendra `control=$Gfillinterior`. Pour dessiner le contour seulement, on prendra `control=$Gborder`. Les quatre derniers arguments de chaque commande définissent deux coins opposés du rectangle.

Dessiner une ellipse

En coordonnées physiques :

```
dummy2 = ellipse (control, x1, y1, x2, y2)
```

En coordonnées réelles:

```
dummy2 = ellipse_w (control, wx1, wy1, wx2, wy2)
```

control vaut à nouveau \$Gfillinterior ou \$Gborder. Les quatre derniers arguments de chaque commande définissent les extrémités de l'ellipse.

Ecrire du texte en mode graphique

Pour écrire du texte en mode graphique, les instructions suivantes sont d'application:

```
integer(kind=2) :: dummy2, x1, x2
```

```
real(kind=8) :: wx1, wx2
```

```
type(wxycoord) :: wxy
```

```
type(xycoord) :: xy
```

```
dummy2 = initializefonts()
```

! initialiser les fonts

```
dummy2 = setfont('t"modern"h14w9')
```

! choisir la font

```
call setcolor(15)
```

! choisir la couleur

```
call moveto (x1, x2, xy)
```

! pos en coord physiques

```
call moveto_w (wx1, wx2, wxy)
```

! pos en coord réelles

```
call outgtext(« I like Fortran 90 »)
```

! écrire



Workspace 'Exemple1': 1 proje
 Exemple1 files
 Source Files
 Exemple1.f90
 Header Files
 Resource Files

Program Main

```

use DFLIB
implicit none
integer(kind=2) :: dummy2, resx = 796, resy = 435
real(kind=8) :: xmin, xmax, ymin, ymax, wx1, wy1, wx2, wy2
type(wxycoord) :: wxy
type(xycoord) :: xy
integer :: i, res

xmin = 0._8 ; xmax = 4._8*3.14159265_8 ; ymin = -1._8 ; ymax = 1._8

call clearscreen($Gclearscreen)           ! clear screen

dummy2 = setcolor(5)                       ! border
dummy2 = rectangle($Gborder,1,1,resx-2,resy-2)
dummy2 = setcolor(1)                       ! interior
dummy2 = rectangle($Gfillinterior,4,4,resx-5,resy-5)

call setviewport(4,4,resx-5,resy-5)        ! define window
dummy2 = setwindow(.true.,xmin,ymin,xmax,ymax)

call setcolor(14)                          ! draw x axis
call moveto_w ( xmin, 0._8, wxy )
dummy2 = lineto_w ( xmax, 0._8 )

wx1 = xmin ; wy1 = sin(wx1) ; res = 200    ! draw function
call moveto_w ( wx1, wy1, wxy )
do i = 1, res
  wx2 = xmin + i * (xmax-xmin) / res
  wy2 = sin(wx2)
  dummy2 = lineto_w ( wx2, wy2 )
enddo

dummy2 = initializefonts()                  ! write title
dummy2 = setfont('t','modern','h14w9')
call moveto ( 4, 4, xy )
call outgtext("y=sin(x)")

end Program Main

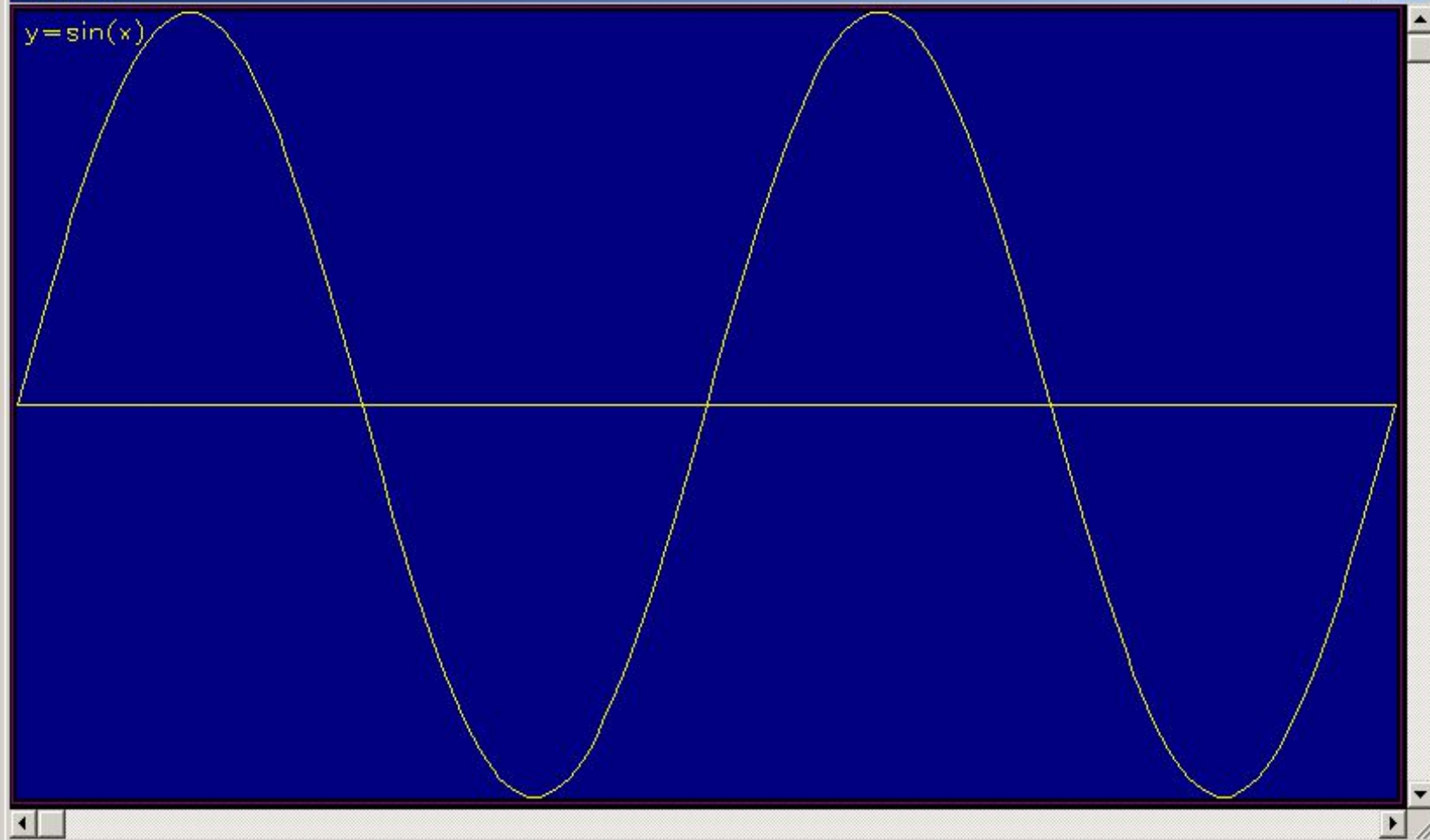
```

FileView

Linking...

Exemple1.exe - 0 error(s), 0 warning(s)

Build Debug Find in Files 1 Find in Files 2





Notions élémentaires d'optimisation

Chapitre 11

Mesurer le temps d'exécution

Program Main

```
real(kind=4) :: time1, time2
```

```
call cpu_time(time1)
```

```
...
```

```
call cpu_time(time2)
```

```
print *, "Temps d'execution (s) : ", time2-time1
```

```
end Program Main
```

Workspace 'Exemple1': 1 proje

- Exemple1 files
 - Exemple1.f90

FileView

```
Program Main

real(kind=4) :: time1, time2
real(kind=8) :: x
integer :: i, j, imax

call cpu_time(time1)

imax = 0
do j = 0, 30
  imax = imax + 2**j
  x = 0._8
  do i = 1, imax
    x = x + i
  enddo
enddo

print *, "Plus grand entier representable : ", imax
print *, "Somme de tous les entiers positifs representables : ", x
print *

call cpu_time(time2)

print *, "Temps d'execution (s) : ", time2-time1

end Program Main
```

```
"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\1-Programmes du cours\Ch...
Plus grand entier representable : 2147483647
Somme de tous les entiers positifs representables : 2.305843007133324E+018

Temps d'execution (s) : 22.16187
Press any key to continue_
```

Comment accélérer un programme ?

Premier grand principe:

choisir les options permettant au compilateur
d'optimiser votre programme

Dans le Developer Studio, aller dans Build\Set Active Configuration... et choisir « Release » plutôt que « Debug ».

Des options plus avancées se trouvent dans
Project\Settings\Fortran\Category:Optimizations



- Compile Exemple1.f90 Ctrl+F7
- Build Exemple1.exe F7
- Rebuild All
- Batch Build...
- Clean
- Update All Dependencies...
- Start Debug
- Debugger Remote Connection...
- Execute Exemple1.exe Ctrl+F5
- Set Active Configuration...**
- Configurations...
- Profile...

Workspace 'Exemple1': 1 proje

- Exemple1 files
 - Exemple1.f90

FileView

```

time2

print *, "Plus grand entier representable : " , imax
print *, "Somme de tous les entiers de 1 a imax : " , sum(1:imax)
print *

call cpu_time(time2)

print *, "Temps d'execution (s) : " , time2

end Program Main

```

Set Active Project Configuration

Project configurations:

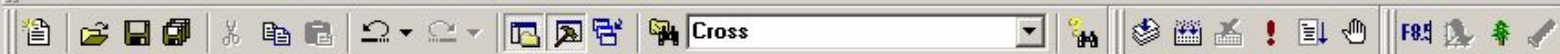
- Exemple1 - Win32 Release
- Exemple1 - Win32 Debug

OK Cancel

-----Configuration: Exemple1 - Win32 Debug-----

Exemple1.exe - 0 error(s), 0 warning(s)

Build Debug Find in Files 1 Find in Files 2



Workspace 'Exemple1': 1 proje

- Exemple1 files
 - Exemple1.f90

FileView

```
Program Main
real(kind=4) :: time1, time2
real(kind=8) :: x
integer :: i, j, imax

call cpu_time(time1)

imax = 0
do j = 0, 30
  imax = imax + 2**j
  x = 0._8
  do i = 1, imax
    x = x + i
  enddo
enddo

print *, "Plus grand entier representable : ", imax
print *, "Somme de tous les entiers positifs representables : ", x
print *

call cpu_time(time2)

print *, "Temps d'execution (s) : ", time2-time1

end Program Main
```

```
"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\1-Programmes du cours\Ch...
Plus grand entier representable : 2147483647
Somme de tous les entiers positifs representables : 2.305843007133324E+018
Temps d'execution (s) : 10.79552
Press any key to continue_
```

```
-----Configuration: Exemple1 - Win32 Release-----
Exemple1.exe - 0 error(s), 0 warning(s)
```

Comment accélérer un programme ?

Premier grand principe:

choisir les options permettant au compilateur
d'optimiser votre programme

Avec un compilateur tel que gfortran, compilez avec
l'option `-O2`.

Comment accélérer un programme ?

Deuxième grand principe:

minimiser le nombre d'opérations à réaliser
nettoyer les boucles
définir des variables intermédiaires

Il faut sortir des boucles les calculs qui peuvent être réalisés en dehors de celles-ci.

Dans le même ordre d'idées, il peut être utile de définir des variables contenant des sommes ou des produits partiels plutôt que reproduire inutilement le calcul de ces quantités.

L'objectif est de réduire le temps CPU.

Workspace 'Exemple1': 1 proje

- Exemple1 files
 - Exemple1.f90

FileView

```

Program Main
  real(kind=4) :: time1, time2
  real(kind=8) :: x
  integer :: i, j, imax

  call cpu_time(time1)

  imax = 0
  do j = 0, 30
    imax = imax + 2**j
  enddo
  x = 0._8
  do i = 1, imax
    x = x + i
  enddo

  print *, "Plus grand entier representable : ", imax
  print *, "Somme de tous les entiers positifs representables : ", x
  print *

  call cpu_time(time2)

  print *, "Temps d'execution (s) : ", time2-time1
end Program Main

```

```

"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortran 90\1-Programmes du cours\Ch...
Plus grand entier representable : 2147483647
Somme de tous les entiers positifs representables : 2.305843007133324E+018

Temps d'execution (s) : 5.608064
Press any key to continue_

```

```

-----Configuration: Exemple1 - Win32 Release-----
Linking...
Exemple1.exe - 0 error(s), 0 warning(s)

```

Comment accélérer un programme ?

Troisième grand principe:

accéder aux éléments d'un tableau
dans leur ordre naturel

efficace

```
do j=1,n  
  do i=1,n  
    u=matrice(i,j)...
```

inefficace

```
do i=1,n  
  do j=1,n  
    u=matrice(i,j)...
```

L'objectif est de réduire le temps d'accès à la mémoire.

Exemple avec un produit matrice-vecteur

n=6000

Debug

Release

```
Ax = 0._r  
do i = 1, n  
  do j = 1, n  
    Ax(i) = Ax(i) + A(i,j) * x(j)  
  enddo  
enddo
```

4.41 sec

1.83 sec

```
Ax = matmul(A,x)
```

3.22 sec

1.82 sec

```
Ax = 0._r  
do j = 1, n  
  do i = 1, n  
    Ax(i) = Ax(i) + A(i,j) * x(j)  
  enddo  
enddo
```

1.05 sec

0.29 sec

```
Ax = 0._r  
do j = 1, n  
  Ax(:) = Ax(:) + A(:,j) * x(j)  
enddo
```

0.79 sec

0.29 sec

Program Main

```
use UtilityPack
```

```
real(kind=4) :: time1, time2
```

```
call cpu_time(time1)
```

```
...
```

```
call cpu_time(time2)
```

```
print *, "Temps d'execution : ", TimeToString(time2-time1)
```

```
end Program Main
```

Temps d'exécution : 3 heures 24 min 26 sec (12265.76 sec)

```
Module UtilityPack
```

```
contains
```

```
Function TimeToString (time)
```

```
real(kind=4),intent(in) :: time
```

```
character(len=50) :: TimeToString
```

```
integer :: sec, min, hour, days, itime
```

```
character(len=20) :: Ssec, Smin, Shour, Sdays
```

```
itime = nint(time) ; sec = itime - (itime/60)*60
```

```
itime = (itime - sec) / 60 ; min = itime - (itime/60)*60
```

```
itime = (itime - min) / 60 ; hour = itime - (itime/24)*24 ; days = (itime - hour) / 24
```

```
write(Ssec,'(i2)') sec ; write(Smin,'(i2)') min ; write(Shour,'(i2)') hour ; write(Sdays,'(i4)') days
```

```
TimeToString = trim(adjustl(Ssec)) // " sec"
```

```
if (min+hour+days>0) TimeToString = trim(adjustl(Smin)) // " min " // TimeToString
```

```
if (hour+days>0) then
```

```
if (hour>1) then
```

```
TimeToString = " heures "// TimeToString
```

```
else
```

```
TimeToString = " heure "// TimeToString
```

```
endif
```

```
TimeToString = trim(adjustl(Shour)) // TimeToString
```

```
endif
```

```
if (days>0) then
```

```
if (days>1) then
```

```
TimeToString = " jours " // TimeToString
```

```
else
```

```
TimeToString = " jour " // TimeToString
```

```
endif
```

```
TimeToString = trim(adjustl(Sdays)) // TimeToString
```

```
endif
```

```
write(Ssec,'(f9.2)') time
```

```
TimeToString = trim(adjustl(TimeToString)) // " (" // trim(adjustl(Ssec)) // " sec)"
```

```
end Function TimeToString
```

```
end Module UtilityPack
```




Exécution en lignes de commandes

Chapitre 12

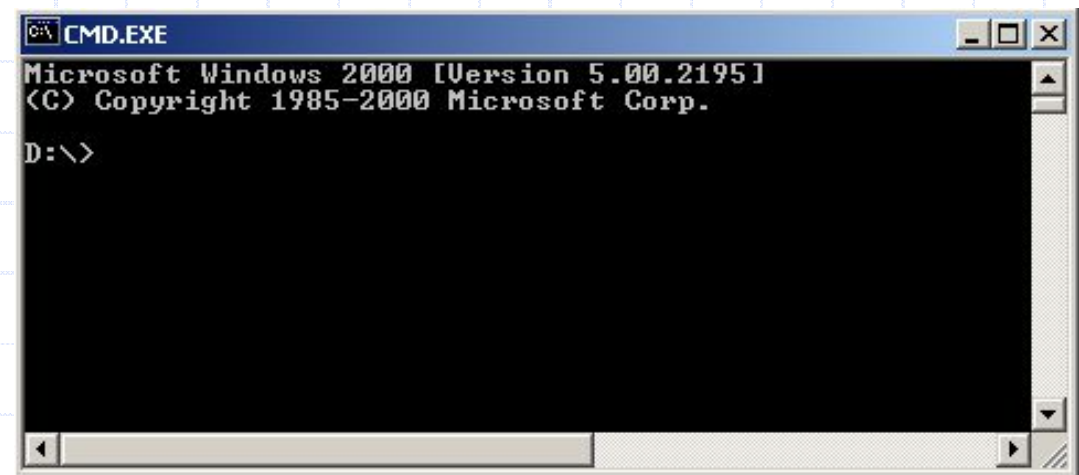
Ouvrir une fenêtre d'invite de commandes

En Windows :

Démarrer\Tous les Programmes\Accessoires\Invite de commandes

ou

Démarrer\Exécuter ... cmd



Instructions de base (Windows)

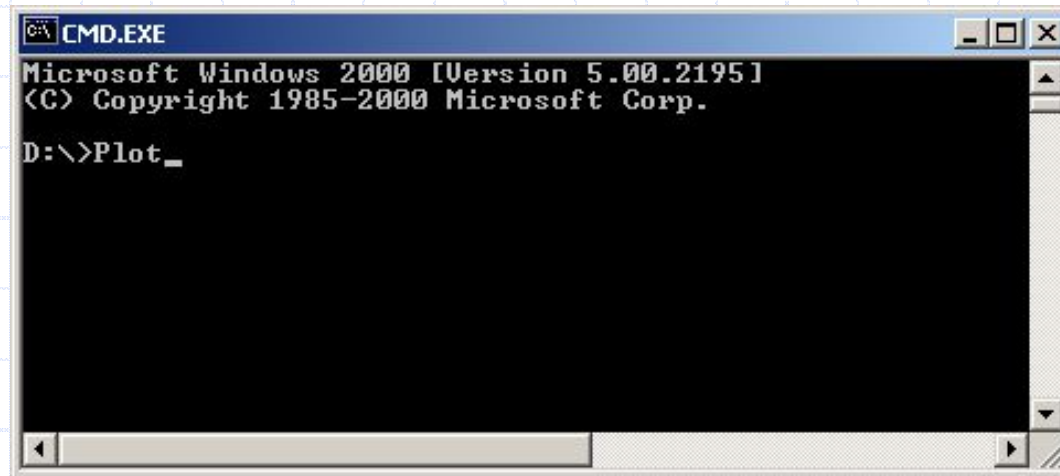
- **dir**: affiche le contenu du répertoire courant
- **c:**, **d:**, ... : changer de disque
- **cd**: changer de répertoire
 - **cd dir1**: aller dans le répertoire dir1
 - **cd dir1\dir2**: aller d'abord dans le répertoire dir1, ensuite dans le répertoire dir2
 - **cd ..**: remonter d'un niveau dans l'arborescence des répertoires
- **mkdir**: créer un répertoire
- **copy**: copier des fichiers
- **move**: déplacer des fichiers ou des répertoires
- **del**: effacer des fichiers
- **help**: aide sur les instructions disponibles et leurs options (exemples: help, help dir)

Instructions de base (Linux)

- **ls**: affiche le contenu du répertoire courant
- **cd**: changer de répertoire
 - `cd dir1`: aller dans le répertoire dir1
 - `cd dir1/dir2`: aller d'abord dans le répertoire dir1, ensuite dans le répertoire dir2
 - `cd ..`: remonter d'un niveau dans l'arborescence des répertoires
- **mkdir**: créer un répertoire
- **cp**: copier des fichiers
- **mv**: déplacer des fichiers ou des répertoires
- **rm**: effacer des fichiers
- **man**: aide sur les instructions disponibles et leurs options (exemples: `man gfortran`)

Exécution en lignes de commandes

On peut lancer un exécutable en lignes de commandes en tapant son nom. Cet exécutable doit se trouver dans le répertoire courant ou dans un répertoire indiqué par la variable d'environnement PATH.



Dans Ubuntu, on lance une invite de commande en allant dans Applications\Utilitaires\Terminal.

Exécution en lignes de commandes

Pour modifier cette variable d'environnement :

Windows XP :

- Poste de Travail \ [bouton droit] \ Propriétés
- Avancé
- Variables d'environnement...
- Sélectionner PATH \ Modifier...

Les répertoires indiqués par la variable d'environnement PATH doivent être séparés par des point-virgules. Vous pouvez ainsi indiquer le répertoire dans lequel vous mettrez tous vos exécutables.

Exécution en lignes de commandes

Pour modifier cette variable d'environnement :

Windows 7 :

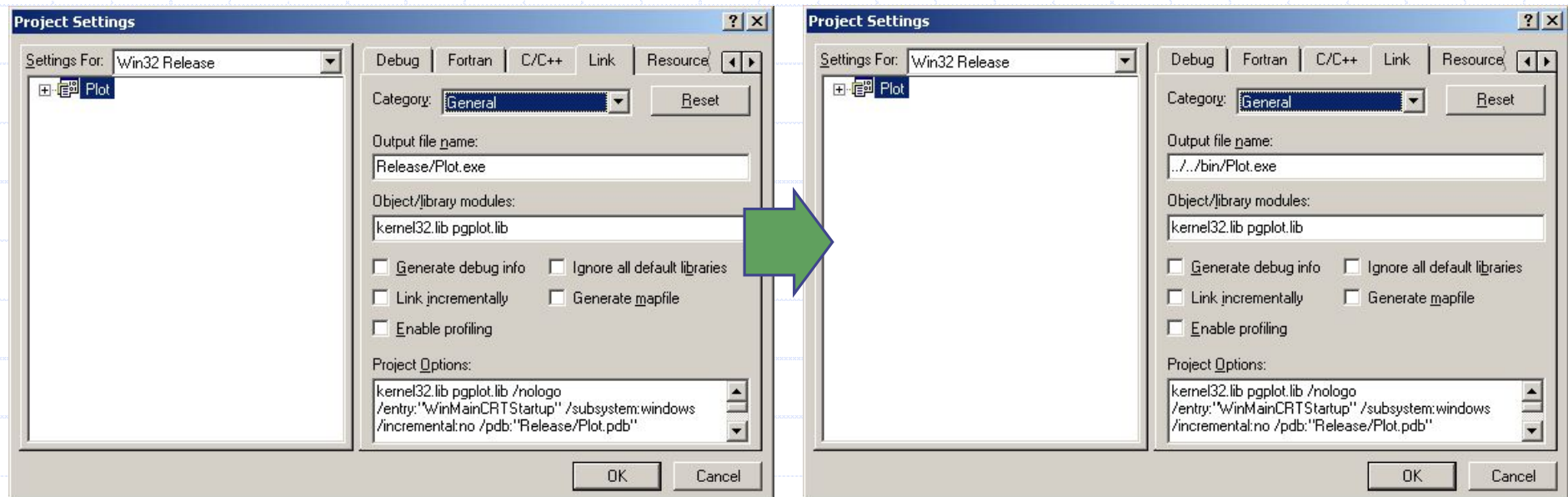
- Démarrer \ Panneau de Configuration
- Système \ Paramètres système avancés
- Variables d'environnement...
- Sélectionner PATH \ Modifier...

Les répertoires indiqués par la variable d'environnement PATH doivent être séparés par des point-virgules. Vous pouvez ainsi indiquer le répertoire dans lequel vous mettrez tous vos exécutables.

Exécution en lignes de commandes

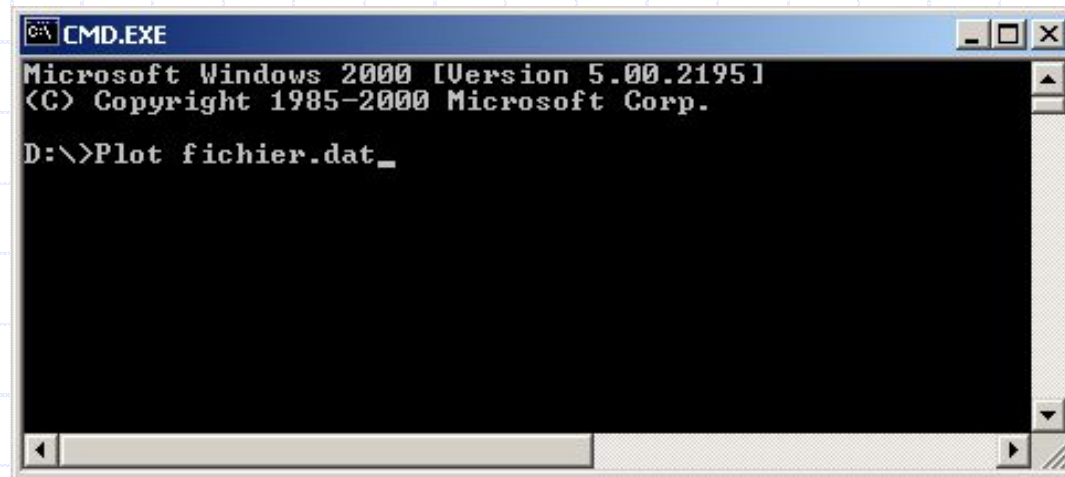
Vous pouvez indiquer dans le Developer Studio le répertoire destiné à recevoir l'exécutable.

Project\Settings...\Link\Output file name:



Exécution avec arguments

Le nom de l'exécutable peut être suivi d'un ou plusieurs arguments.



```
C:\> CMD.EXE
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

D:\>Plot fichier.dat_
```

Le programme peut récupérer ces arguments grâce aux instructions `nargs` et `getarg`.

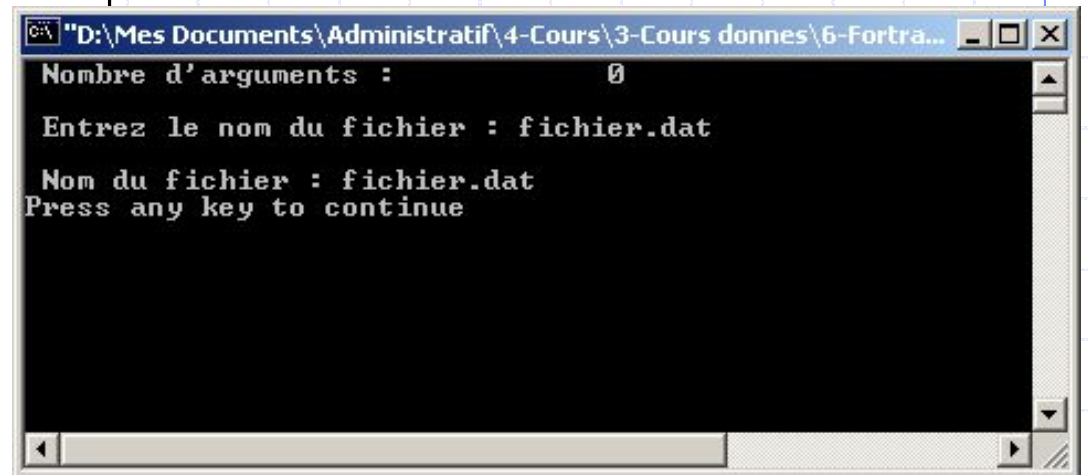
Récupération des arguments

```
Program Main
integer :: narg
character(len=50) :: name

narg = nargs()-1
print *, "Nombre d'arguments : ", narg
print *

if (narg>0) then
  call getarg(1,name)
else
  write(*,'(x,a,$)') "Entrez le nom du fichier : "
  read(*,'(a)') name ; print *
endif

print *, "Nom du fichier : ", name
end Program Main
```



```
"D:\Mes Documents\Administratif\4-Cours\3-Cours donnees\6-Fortra...
Nombre d'arguments :          0
Entrez le nom du fichier : fichier.dat
Nom du fichier : fichier.dat
Press any key to continue
```



```
CMD.EXE
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

D:\>Plot fichier.dat
Nombre d'arguments :          1
Nom du fichier : fichier.dat

D:\>_
```

Exécution de commandes en Fortran

```
call System(« dir »)
```

```
call System(« dir > fichier.dat »)
```

L'instruction `System` permet d'exécuter des commandes DOS en Fortran (pour un environnement Windows).

Elle permet également d'exécuter des commandes Linux si on est dans ce type d'environnement.

```
call System(« ls > fichier.dat »)
```

Download

Vous pouvez retrouver ce document sur

<http://perso.unamur.be/~amayer>