Programmer en Fortran 90

Cédric Vandenbem

Laboratoire de Physique du Solide
Centre de Recherche en Physique de la Matière et Rayonnement
cedric.vandenbem@fundp.ac.be

http://perso.fundp.ac.be/~cvandenb/

MCours.com

Table des matières

Références

- Livres :
  - C. Delannoy, *Programmer en Fortran 90 – Guide Complet*, 2nd Edition (Eyrolles, 1997)
  - S. J. Chapman, *Fortran 95/2003 for Scientists and Engineers*, 3rd Edition (McGraw Hill, 2007)
  - N. Dubesset et J. Vignes, *Le Fortran – Le langage normalisé* (Technip, 1991)

- Cours :
  - A. Mayer, *Cours de programmation : Fortran 90*, http://perso.fundp.ac.be/~amayer

Chapitre 1. Généralités sur le Fortran 90

Exemple de programme en Fortran 90

```
!*****************************************************************************
!
!  PROGRAM: racines_carrees.f90
!
!  PURPOSE: Calcul de nrac racines carrées demandées de manière interactive
!
!*****************************************************************************

program racines_carrees

    implicit none

    integer :: i,nrac=3
    real :: valeur,racine

    print *, 'Bonjour - Calculons ', nrac,' racines carrees'

    do i = 1,nrac
       print *, 'Donnez un nombre :'
       read *, valeur
       if (valeur.ge.0) then
          racine = sqrt(valeur)
          print *,'Le nombre ', valeur, 'a pour racine : ', racine
       else
          print *,'Le nombre ', valeur, 'ne possede pas de racine'
       end if
    end do
    print *, 'Job termine - Au revoir'

end program racines_carrees
```

```
"D:\COURS\PROGRAMMATION_BAC2\PROGRAMMES\Exemple\Debug\E
 Bonjour - Calculons             3  racines carrees
 Donnez un nombre :
25
 Le nombre    25.00000      a pour racine :    5.000000
 Donnez un nombre :
-3.58
 Le nombre   -3.580000      ne possede pas de racine
 Donnez un nombre :
1.25e4
 Le nombre    12500.00      a pour racine :    111.8034
 Job termine - Au revoir
Press any key to continue
```

Structure d'un programme (1)

```
program <Nom du programme>
  :
  : Instructions de déclaration
  :


  :
  : Instructions exécutables
  :
end program <Nom du programme>
```

- Instructions de déclaration:
  - Déclaration des objets utilisés
  - Informations utiles à la compilation
  - Toujours précéder les instructions exécutables
- Instructions exécutables:
  - Opérations mathématiques
  - Lecture/écriture de fichiers
  - Appel à des sous-programmes
  - …

Structure d'un programme (2)

```
!*****************************************************************************
!
!  PROGRAM: racines_carrees.f90
!
!  PURPOSE: Calcul de nrac racines carrées demandées de manière interactive
!
!*****************************************************************************

program racines_carrees

   implicit none

   integer :: i,nrac=3
   real :: valeur,racine

   print *, 'Bonjour - Calculons ', nrac,' racines carrees'

   do i = 1,nrac
      print *, 'Donnez un nombre :'
      read *, valeur
      if (valeur.ge.0) then
         racine = sqrt(valeur)
         print *,'Le nombre ', valeur, 'a pour racine : ', racine
      else
         print *,'Le nombre ', valeur, 'ne possede pas de racine'
      end if
   end do
   print *, 'Job termine - Au revoir'

end program racines_carrees
```

Instructions de déclaration

Instructions exécutables

Règles d'écriture : les noms de variables

- Ils désignent les différents « objets » manipulés par le programme
- Ils sont composés d'une suite de caractères alphanumériques dont
  - le **premier est une lettre**
  - et limité à **31 caractères**

- En ce qui concerne les lettres:
  - Le caractère souligné (_) est considéré comme une lettre

```
valeur_totale _total valeur_2 _8
```

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

```
Racine ⇔ racine
'Racine' ≠ 'racine'
```

Règles d'écriture : Format libre (1)

- Longueur des lignes : 132 caractères
- Des espaces peuvent être introduits entre les éléments de langage

```
integer::n,resultat,p1
                    ⇔
integer :: n, resultat, p1
```

- Les instructions multiples
  - La fin de ligne sert de séparation naturelle entre deux instructions

```
print *, 'donnez un nombre'
read *, n
```

  - Mais, il est possible de placer plusieurs instructions sur une même ligne en les séparant par ;

```
print *, 'donnez un nombre' ;read *, n
```

Règles d'écriture : Format libre (2)

- Instructions sur plusieurs lignes
  - Une ligne terminée par & se poursuit sur la ligne suivante.
  - Le & peut être reproduit sur la ligne suivante.

```
integer::n,resultat
                    ⇔
integer :: n, &
           resultat
                    ⇔
integer :: n, &
           & resultat
```

  - Pour les chaînes de caractères, il **doit** être répété!

```
print *, 'Hello world'
                    ⇔
print *, 'Hello &
          &world'
```

Règles d'écriture : Format libre (3)

- Commentaires
  - Le ! signifie que le reste de la ligne est considéré comme un commentaire

```
!*****************************************************************************
!
!  PROGRAM: racines_carrees.f90
!
!  PURPOSE: Calcul de nrac racines carrées demandées de manière interactive
!
!*****************************************************************************

program racines_carrees

   implicit none

   integer :: i,nrac=3
   real :: valeur,racine

   print *, 'Bonjour - Calculons ', nrac,' racines carrees'

   do i = 1,nrac
      print *, 'Donnez un nombre :'
      read *, valeur
      if (valeur.ge.0) then      ! cas valeur positive ou nulle
         racine = sqrt(valeur)
         print *,'Le nombre ', valeur, 'a pour racine : ', racine
      else                       ! cas valeur négative
         print *,'Le nombre ', valeur, 'ne possede pas de racine'
      end if
   end do
   print *, 'Job termine - Au revoir'

end program racines_carrees
```

Règles d'écriture : Format fixe (1)

- Intérêt :
  - Compatibilité avec des anciens programmes.
- Format :
  - Le format fixe correspond au format du fortran 77 avec 2 extensions:
    - Les instructions multiples
    - La nouvelle forme de commentaires
  - Format Fortran 77:
    - Une ligne contient 72 caractères (au-delà, ce n'est pas considéré par le compilateur)
    - Découpage en zone:
      - Zone étiquette (colonne 1 à 5)
      - Zone instruction (colonne 7 à 72)
      - Colonne suite (colonne 6)
    - Un commentaire commence par C en colonne 1

Format fixe : Extension du fichier .f

Format libre : Extension du fichier .f90

Règles d'écriture : Format fixe (2)

```
c*******************************************************************************
c
c  PROGRAM: racines_carrees.f
!
!  PURPOSE: Calcul de nrac racines carrées demandées de manière interactive
!
!*******************************************************************************

      program racines_carrees

      implicit none

      integer :: i,nrac=3
      real :: valeur,racine

      print *, 'Bonjour - Calculons ', nrac,' racines carrees'

      do i = 1,nrac
         print *, 'Donnez un nombre :'
         read *, valeur
         if (valeur.ge.0) then      ! cas valeur positive ou nulle
            racine = sqrt(valeur)
            print *,'Le nombre ', valeur, 'a pour racine : ',
     &           racine
         else                       ! cas valeur négative
            print *,'Le nombre ', valeur, 'ne possede pas de racine'
         end if
      end do
      print *, 'Job termine - Au revoir'

      end program racines_carrees
```

Règles d'écriture : Format fixe (3)

- La notion de séparateur n'existe plus
  - L'espace est sans signification

```
doi = 1, nrac
                    ⇔
d   oi = 1,n rac
                    ⇔
do i = 1,nrac
```

- Incompatibilités des deux formats
  - Il faut préciser, lors de la compilation, le format employé.
  - Il est possible de faire de la « compilation séparée », très utile pour récupérer des routines écrites en Fortran 77.

Chapitre 2. Les types de données

Les différents types

- Integer :
  - Nombres entiers
  - Valeurs positives ou négatives entre des limites qui dépendent de la machine
- Real :
  - Nombres réels
  - Valeurs positives ou négatives, ayant une partie fractionnaire, entre des limites qui dépendent de la machine
- Complex :
  - Nombres complexes composés d'une partie réelle et d'une partie imaginaire, toutes deux de type réel
- Logical
  - Valeurs booléennes ou logiques ne prenant que deux valeurs: vrai (*true*) ou faux (*false*)
- Character :
  - Chaînes d'un ou plusieurs caractères

Le type INTEGER

- Représentation des nombres entiers relatifs (*complément à 2*)

- Limite en fonction de la machine
  - 16 bits (2 octets): -32.768 ≤ i ≤ 32.767
  - 32 bits (4 octets): -2.147.483.648 ≤ i ≤ 2.147.483.647

- Pour introduire une constante entière, il suffit de l'écrire sous sa forme décimale habituelle avec ou sans signe:

```
+533        48      -2894
```

- Il est également possible de l'écrire en base 2, 8 ou 16:

```
B'0110101110'

O'025472'

Z'FA0FA'
```

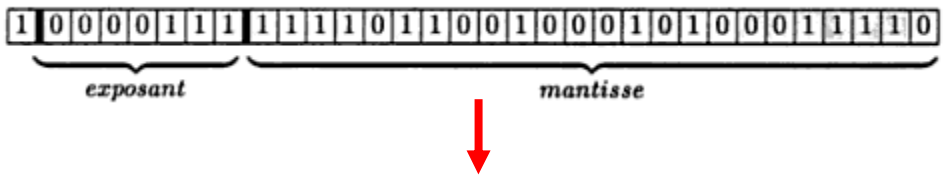
Le type REAL : Représentation en mémoire

- Virgule flottante normalisée
  - Le nombre A se décompose sous la forme: $A = m*b^c$ tel que $1/b \leq m < 1$
    - m est la mantisse (nombre algébrique à partie entière et partie décimale)
    - b est la base du système de numérotation (système binaire)
    - c est l'exposant ou caractéristique (nombre algébrique entier)

-123.135 =	-0.123135	10^3
+84.0 =	+0.84	10^2

- Erreur de troncature
  - Utilisateur : entrées/sorties en base décimale
  - Ordinateur : travail en base binaire

$$-123.135_{(10)} = -0.11110110010001010001111010 * 10^{111}{}_{(2)}$$

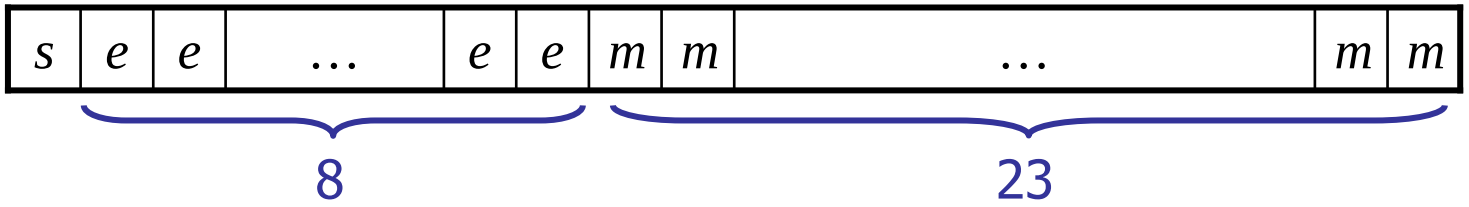


$$-123.1349945068359_{(10)}$$

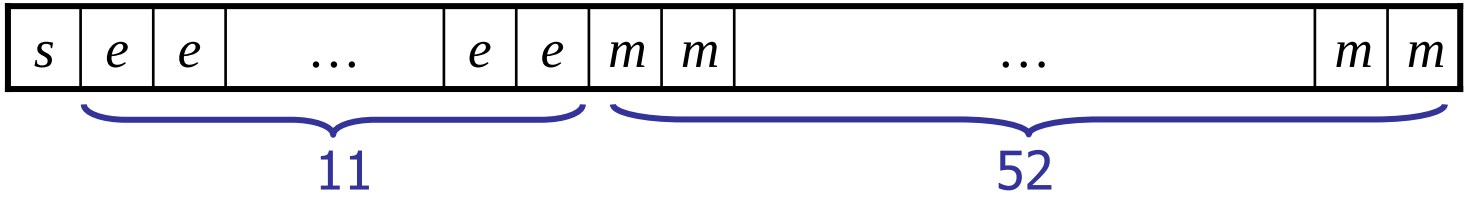
Le type REAL donne une représentation approchée des nombres réels!

Le type REAL : Limite de représentation

- Réel simple précision
  - Motif binaire de la forme :



  - Nombre représenté : $r = s1.m\ 2^{e-127}$
  - 32 bits (4 octets): $1.2\ 10^{-38} \leq |x| \leq 3.4\ 10^{38}$
  - Précision : 7 chiffres significatifs
- Réel double précision
  - Motif binaire de la forme:



  - Nombre représenté : $r = s1.m\ 2^{e-1023}$
  - 64 bits (8 octets): $2.2\ 10^{-308} \leq |x| \leq 1.8\ 10^{308}$
  - Précision : 16 chiffres significatifs

Le type REAL : Notation des constantes

- Les constantes de type réel s'écrivent indifféremment suivant l'une des deux notations:
  - Décimale (doit comporter obligatoirement un .)

```
12.43   -0.38   .38   4.   -.27
```

  - Exponentielle (utilisation de la lettre e, le . peut être absent)

```
4.25E4    4.25e+4     425e2
0.54e-2                5.4e-3
48e13     48.e13     48.0e13
```

- En double précision (codage 8 octets), les constantes s'écrivent obligatoirement sous forme exponentielle en utilisant la lettre d à la place de e.

```
4.25D4    4.25d+4     425d2
```

Les types implicites

- Fortran 90 ne rend pas obligatoire les déclarations des types des variables. Fortran attribue implicitement un type à une variables non définies en fonction de la première lettre:
  - I, J, K, L, M, N représentent des types INTEGER
  - Les autres représentent des types REAL

- Plus néfaste qu'utile

```
integer :: nbre = 5, k

……

k = nbr + 1
```

nbr au lieu de nbre ne sera pas détecté par le compilateur

- Solution : Supprimer les types implicites

```
implicit none
```

Le Type COMPLEX

- Pour Fortran, un nombre complexe est un couple de nombres réels.

- En mémoire, le type *complex* est représenté par deux nombres de type *real*.

- Les constantes de type *complex*

```
( partie_reelle, partie_imaginaire )
```

```
(2.5,3.e2)  2,5 + 300i
(0.,1.)             i
```

- L'assignation

```
variable = cmplx(variable_r,variable_i)
```

```
complex :: z
real :: x,y
……
z = cmplx(x,y)
```

Le type LOGICAL

```
program exemple_logique1

   implicit none

   integer :: n,p

   print *, 'donnez 2 nombres entiers'
   read *, n, p
   if (n<p) then
      print *, 'croissant'
   else
      print *, 'non croissant'
   end if

end program exemple_logique1
```

```
"D:\Cours\Programmation_Bac2\Programmes\Ex
 donnez 2 nombres entiers
25 10
 non croissant
Press any key to continue_
```

```
program exemple_logique2

   implicit none

   integer :: n,p
   logical :: range

   print *, 'donnez 2 nombres entiers'
   read *, n, p
   range = n<p
   if (range) then
      print *, 'croissant'
   else
      print *, 'non croissant'
   end if

end program exemple_logique2
```

```
"D:\Cours\Programmation_Bac2\Programmes\Exer
 donnez 2 nombres entiers
25 10
 non croissant
Press any key to continue_
```

```
program exemple_logique3

   implicit none

   integer :: n,p
   logical :: range

   print *, 'donnez 2 nombres entiers'
   read *, n, p
   range = n<p
   print *, 'n<p?', range

end program exemple_logique3
```

```
"D:\Cours\Programmation_Bac2\Programmes\E
 donnez 2 nombres entiers
25 10
 n<p? F
Press any key to continue_
```

Déclaration des variables numériques et logiques (1)

```
<type> [ [,<attribut(s)>] :: ] <variable> [=<value>]
```

- Type:
  - INTEGER[([KIND=] variante)]
  - REAL[([KIND=] variante)]
  - COMPLEX[([KIND=] variante)]
  - LOGICAL[([KIND=] variante)]
  - DOUBLE PRECISION

- Attribut:
  - parameter, public, private, pointer, target, allocatable, optional, save, external, intrinsic, intent, dimension

- Si <attribut(s)> ou =<value>, le :: est obligatoire

Déclaration des variables numériques et logiques (2)

```
integer :: nbre = 5, k
real a
real :: c,d,e
interger, parameter :: maxat=100
integer,parameter :: at_xyz=3*maxat
real,dimension(3) ::  a_vec,b_vec
logical :: converged=.FALSE.
complex(kind=4) :: z=(1.0,0.1)
```

Les expressions constantes

- Fortran permet de définir une « constante symbolique », un symbole auquel on attribue une valeur qui ne pourra être modifiée par le programme

```
<type>, parameter :: <variable(s)> = <value>
```

```
interger, parameter :: n1=50, n2=120
integer,parameter :: nbv=abs(n1-n2)+1
```

- Conseillé:
  - si la variable ne change pas lors de l'exécution du programme
  - pour rendre un code plus lisible (pi, deux_pi, …)
  - pour rendre un code plus facile à modifier (dimension d'une matrice)

Les types de données paramétrisés (1)

- Ils assurent la portabilité de la précision numérique

- Les compilateurs supportent au moins deux sortes de réels, identifiés par un nombre entier (nombre d'octets utilisés pour la représentation de ces réels)

```
real(kind=4) :: x !simple précision
real(kind=8) :: y !double précision
```

- Le kind de constantes littérales se note par exemple 3.5_8

- RQ : tous les compilateurs ne codent pas forcément la simple et la double précision avec kind=4 et kind=8

Les types de données paramétrisés (2)

- Pour faciliter la conversion de tout un programme d'un kind vers un autre, on définit

```
integer,parameter :: r=4
real(kind=r) :: x,y=2._r
```

- Détermination « propre » d'un kind:
  - selected_int_kind(r): valeur de la première sorte d'entiers disponibles capables de représenter les entiers de -10^r à 10^r (r=*range*)
  - selected_real_kind(p,r): valeur de la première sorte de réels disponibles capables de représenter les réels ayant au moins p chiffres décimaux significatifs (p=*precision*) de -10^r à 10^r (r=*range*)
  - Si la précision n'est pas disponible, les fonctions retournent la valeur -1

Les types de données paramétrisés (3)

- Exemple:
  - x et y ont 15 chiffres significatifs et peuvent prendre des valeurs comprises entre -10^{70} et 10^{70}

```
integer,parameter :: r=selected_real_kind(15,70)
real(kind=r) :: x,y=2._r
```

Déclaration du Type CHARACTER

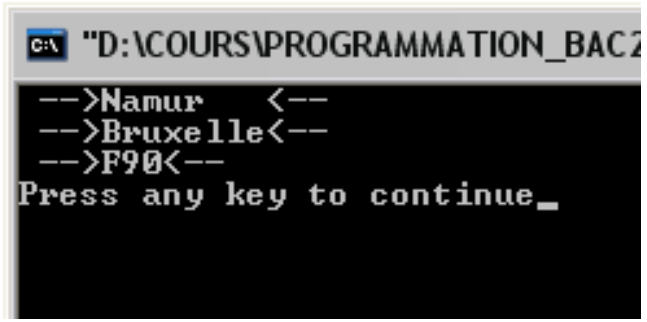
```
character[(len=<longueur>)] [,<attribut(s)>] [::] <variable> [=<value>]
```

- Longueur:
  - Expression entière ou * (déduction par le compilateur de la longueur)

```
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
```

Le type CHARACTER : Exemples

```
program Exemple_character

   implicit none

   character(len=8) :: ville     = 'Namur'       !Padded right
   character(len=8) :: capitale = 'Bruxelles'   !Truncated
   character(len=*),parameter :: lang = 'F90'   !Assumed length

   print *, '-->'//ville//'<--'
   print *, '-->'//capitale//'<--'
   print *, '-->'//lang//'<--'

end program Exemple_character
```



Chapitre 3. Les expressions et l’instruction d’affectation

Introduction

- L'instruction d'affectation se présente sous la forme:

```
<variable> = <expression>
```

- Expression peut faire intervenir
  - des opérateurs comme la division entière, l'exponentiation,…
  - plusieurs opérateurs et, donc, des problèmes de priorité
  - des variables de types différents, ce qui engendre des conversions implicites
  - une variable ayant un type différent du résultat fourni par l'expression engendre des conversions forcées

La valeur d'une expression est entièrement évaluée avant d'être affectée

Expressions arithmétiques : Opérateurs usuels

- Quatre opérateurs dyadiques :
  - Addition : +
  - Soustraction : -
  - Multiplication : *
  - Division : / (Pas de division par zéro!)

- Un opérateur monadique :
  - Opposé : -

Le résultat de la division entre deux entiers est un entier (partie entière du quotient exact).

```
5/2 = 2     7/5 = 1     -8/3 = -2
```

Expressions arithmétiques : Opérateur d'exponentiation

- Il se note : **
- Il correspond à l'opération mathématique a^b = a**b
- Pour r entier positif:
  - x^r = x*x*x…*x (r fois)
  - $x^{-r} = 1/x^r$
- Pour r réel quelconque:
  - $x^r = e^{r \text{ Log } x}$ pour $x \geq 0$
  - Ce n'est qu'à l'exécution qu'on obtient un message d'erreur si x < 0!

Expressions arithmétiques : Priorités relatives

- Ordre des opérations
  - Exponentiation
  - Multiplication et division (gauche à droite)
  - Addition et soustraction
- En cas de doute, *utiliser les parenthèses!*

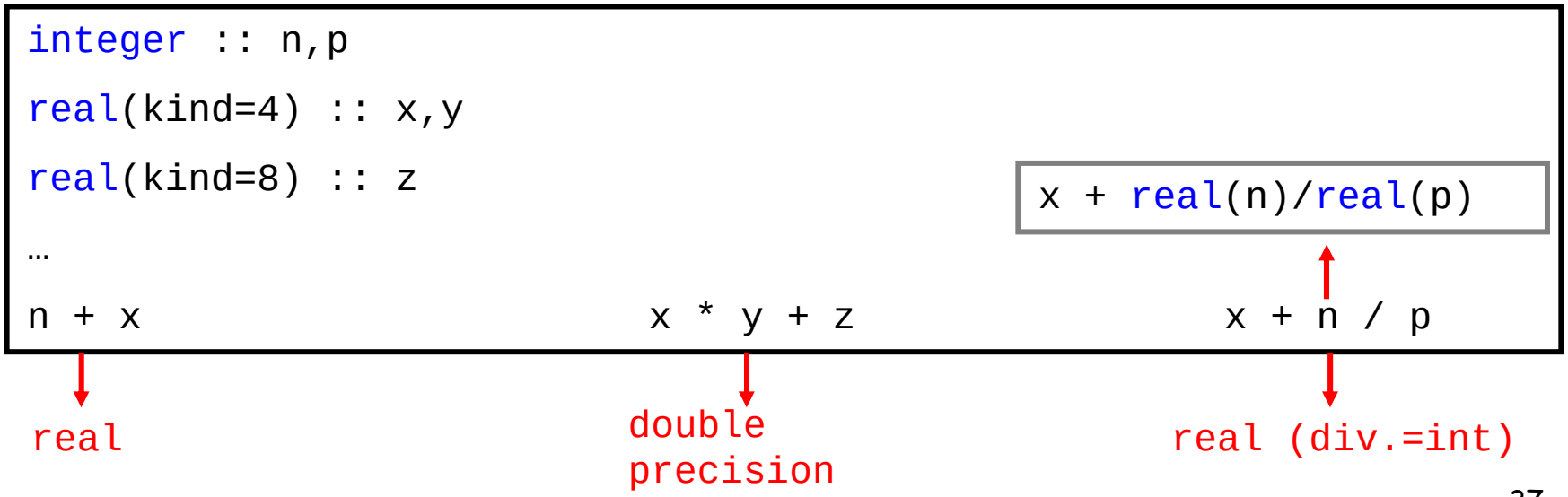
Expressions arithmétiques : Conversion implicite

- Fortran autorise que les opérations dyadiques (+,-,*,/) portent sur des opérandes de types différents.

  ⇒ Le compilateur met en place des conversions implicites

- Respect de la hiérarchie:

```
integer -> real -> complex

simple precision -> double precision
```



Expressions arithmétiques : Conversion forcée

- Fortran accepte que la variable soit d'un type différent que celui de l'expression (<variable> = <expression>)

  ⇒ Conversion de l'expression dans le type de la variable

- Hiérarchie non respectée

```
integer :: n=5,p=10,q
real(kind=4) :: x=0.5,y=1.25
…
y = n + p                    p = x + y                    y = x / p
```

Nombres complexes : Opérateurs usuels

```
complex(kind=4) :: z
real(kind=4) :: x=0.,y=1.
…
z = cmplx(x,y)                    !assignation simple précision (SP)
…
real(z)                           !partie réelle
imag(z)                           !partie imaginaire
conjg(z)                          !complexe conjugué
abs(z)                            !module
```

Anciens noms des fonctions intrinsèques

- En Fortran 90, beaucoup de procédures sont génériques (un même nom peut désigner plusieurs procédures)

- La fonction abs est générique (dépend de la variable). Dans les anciennes versions de Fortran, elle s'écrit

Nom spécifique	Type de l'argument	Type du résultat
iabs	INTEGER(4)	INTEGER(4)
abs	REAL(4)	REAL(4)
dabs	REAL(8)	REAL(8)
cabs	COMPLEX(4)	REAL(4)
cdabs	COMPLEX(8)	REAL(8)

- Règle :
  - a (ou rien) *real*, d *double precision*, i *integer*, c *complex*

Expressions logiques : les comparaisons (1)

- Comparaison des expressions numériques avec une valeur logique pour résultat

Ancienne notation	Nouvelle notation	signification
.LT.	<	inferieur à
.LE.	<=	inférieur ou égal à
.GT.	>	supérieur à
.GE.	>=	supérieur ou égal à
.EQ.	==	égal à
.NE.	/=	différent de

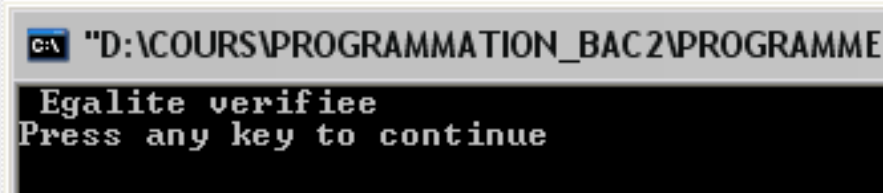
- Seuls == et /= s'appliquent à deux opérandes complexes.

Expressions logiques : les comparaisons (2)

- Encore et toujours la troncature…

```
program exemple_comparaison

   implicit none

   real(kind=8) :: eps=1.e-15

   if (1.0+eps.eq.1.0) then
       print *, 'Egalite verifiee'
   else
      print *, 'Egalite non verifiee'
   end if

end program exemple_comparaison
```


```
program exemple_comparaison

   implicit none

   real(kind=8) :: eps=1.e-16

   if (1.0+eps.eq.1.0) then
       print *, 'Egalite verifiee'
   else
      print *, 'Egalite non verifiee'
   end if

end program exemple_comparaison
```



Expressions logiques : Opérateurs logiques

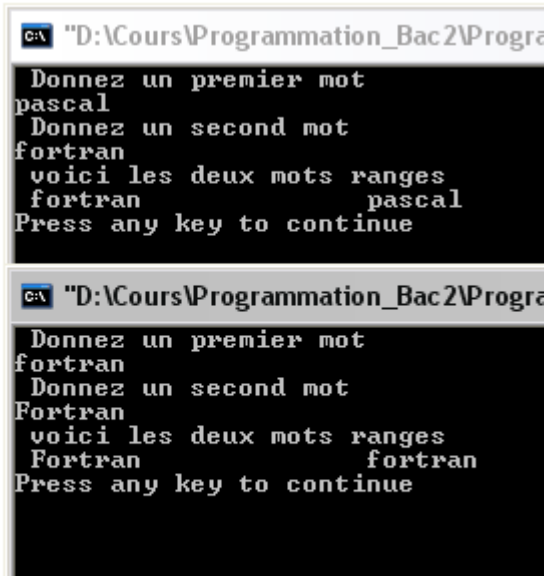
p	q	.not.p	p.and.q	p.or.q	p.eqv.q	p.neqv.q
t	t	f	t	t	t	f
t	f	f	f	t	f	t
f	t	t	f	t	f	t
f	f	t	f	f	t	f

Chaînes de caractères : Opérateurs intrinsèques

- Il existe deux types d'opérateurs:
  - Concaténation (mise bout-à-bout de deux chaînes) : //
  - Comparaison

- Fortran distingue les minuscules des majuscules (ordre voir code ASCII).

```
program Exemple_character2

   implicit none

   character(len=20) :: mot1,mot2,mot

   print *, 'Donnez un premier mot'
   read *, mot1

   print *, 'Donnez un second mot'
   read *, mot2

   if (mot1.gt.mot2) then
      mot = mot1
      mot1 = mot2
      mot2 = mot
   end if

   print *,'voici les deux mots ranges'
   print *, mot1,' ',mot2

end program Exemple_character2
```



Chaînes de caractères : Codage ASCII

- Conversion entier/caractère :

```
<character> = achar(<entier>)
<entier> = iachar(<character>)
```

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Chaînes de caractères : Sous-chaînes (1)

- Manipulation d'une partie formée de caractères consécutifs d'une chaîne:

```
Variable_chaîne([debut]:[fin])
```

  - *debut* et *fin* : expression de type integer
  - Si *debut* absent: 1 par défaut
  - Si *fin* absent : longueur de la chaîne par défaut

```
mot = 'bonjour'
print *, mot(2:6)      → onjou
print *, mot(:3)       → bon
print *, mot(4:)       → jour
mot(2:2) = 'a'
print *, mot(1:5)      → banjo
```

Chaînes de caractères : Sous-chaînes (2)

- Exemples
  - Supprimer la première lettre du mot

```
mot(1:lmot-1) = mot(2:lmot)
mot(lmot:lmot) = ' '
```

  - Afficher le caractère correspondant à un chiffre

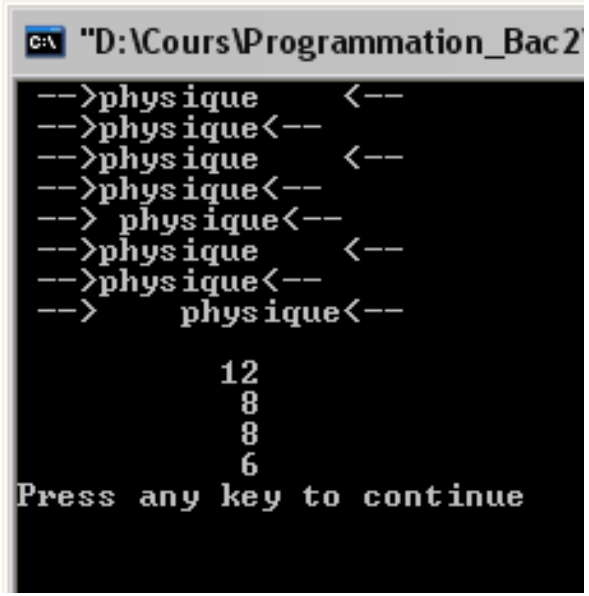
```
character(len=10), parameter :: chiffres = ‘0123456789’
integer :: i
…
print *, chiffre(i+1,i+1)
```

Chaînes de caractères : Autres opérations (1)

```
trim(<variable_chaîne>)                 !suppression blancs de fin
len(<variable_chaîne>)                  !longueur de variable_chaîne
len_trim(<variable_chaîne>)             !equivalent à len(trim(<v_c>)))
trim(adjustl(<variable_chaîne>))        !cadrage à gauche (r à droite)+
                                        !suppression blancs de fin
index(<var1_chaîne>,<var2_chaîne>)      !recherche une sous-chaîne
```

Chaînes de caractères : Autres opérations (2)

```
program Exemple_character3

   implicit none

   character(len=12) :: mot='physique'
   character(len=12) :: mot1=' physique'

   print *, '-->'//mot                    //'<--'
   print *, '-->'//trim(mot)              //'<--'
   print *, '-->'//adjustl(mot)           //'<--'
   print *, '-->'//trim(adjustl(mot))     //'<--'
   print *, '-->'//trim(mot1)             //'<--'
   print *, '-->'//adjustl(mot1)          //'<--'
   print *, '-->'//trim(adjustl(mot1))    //'<--'
   print *, '-->'//adjustr(mot)           //'<--'
   print *
   print *, len(mot)
   print *, len_trim(mot)
   print *, len(trim(mot))
   print *, index(mot,'que')

end program Exemple_character3
```



Chapitre 4. Les instructions de contrôle

Introduction

- L'intérêt d'un programme provient essentiellement de la possibilité d'effectuer :
  - des choix (comportement différent suivant les circonstances)
  - des boucles (répétition d'un ensemble donné d'instructions)

- Fortran 90 est un langage structuré disposant des structures suivantes :
  - choix simple et alternatives « en cascade » avec l'instruction if
  - choix multiple avec l'instruction select case
  - répétition avec l'instruction do.

- La notion de branchement n'est pas absente du Fortran 90 via
  - les branchements inconditionnels : exit, cycle et goto

MCours.com

L'instruction IF : Choix simple

```
if (<exp_logique>) then
 …      ! instructions exécutées si exp_logique est vraie
else
 …      ! instructions exécutées si exp_logique est fausse
end if
```

- Remarques:
  - la partie introduite par le mot clé else peut ne pas exister
  - on peut écrire end if ou endif
  - on peut donner un nom à l'instruction if (imbrications longues)

```
ordre: if (a.lt.b) then
          print *, 'croissant'
else ordre
          print *, 'non croissant'
end if ordre
```

  - on peut imbriquer plusieurs blocs if

L'instruction IF : Alternatives en cascade

```
if (<exp_log1>) then
…   ! instr. exéc. si exp_log1 est vraie
else if (<exp_log2>) then
…   ! instr. exéc. si exp_log1 est fausse et exp_log2 est vraie
else if (<exp_log3>) then
…   ! instr. exéc. si exp_log1, exp_log2 sont fausses et exp_log3 est
    ! vraie
else
…   ! instr. exéc. si exp_log1, exp_log2 et exp_log3 sont fausses
end if
```

L'instruction IF : Cas particulier d'alternative

- Simplification de l'instruction if si
  - pas de *partie else*
  - la *partie then* ne comporte qu'une seule instruction

```
if (x.gt.max) then
        max = x
end if
                ⇔
if (x.gt.max) max = x
```

L'instruction IF : Synthèse (1)

```
[<nom> :] if (<exp_log>) then
              bloc
         [ else if (<exp_log>) then [<nom>]
              bloc
         ]…
         [ else [<nom>]
              bloc
         ]
           end if [<nom>]
```

- exp_log : expression de type LOGICAL
- nom : identificateur
- bloc : ensemble d'instructions simples et/ou structurées

L'instruction IF : Synthèse (2)

```
if (<exp_log>) inst_simple
```

- exp_log : expression de type LOGICAL
- inst_simple : instruction simple exécutable différente d'un autre *if simple* ou de *end*

L'instruction SELECT CASE : Exemple introductif

- Instruction utile pour choisir entre différents blocs d'instructions, en fonction de la valeur d'une expression.

```
integer :: i
…
select case(i)
   case(0)
   …  !s'exécute si i=0
   case(:-1)
   …  !s'exécute si i<=-1
   case(1,5:10)
   …  !s'exécute si i=1 ou 5<=i<=10
   case default
   … !s'exécute dans tous les autres cas
end select
```

Il faut éviter tout recouvrement entre les différentes possibilités proposées!

L'instruction SELECT CASE : Synthèse

```
[<nom> :] select case (<exp_scal>)
        [   case (<sélecteur>) [<nom>]
             bloc
        ]…
        [  case default
             bloc
        ]
          end select [<nom>]
```

- exp_scal : expression scalaire de type INTEGER, LOGICAL, CHARACTER
- sélecteur : liste composée de 1 ou plusieurs éléments de la forme
  - valeur
  - intervalle de la forme [valeur1]:valeur2 ou valeur1:[valeur2]

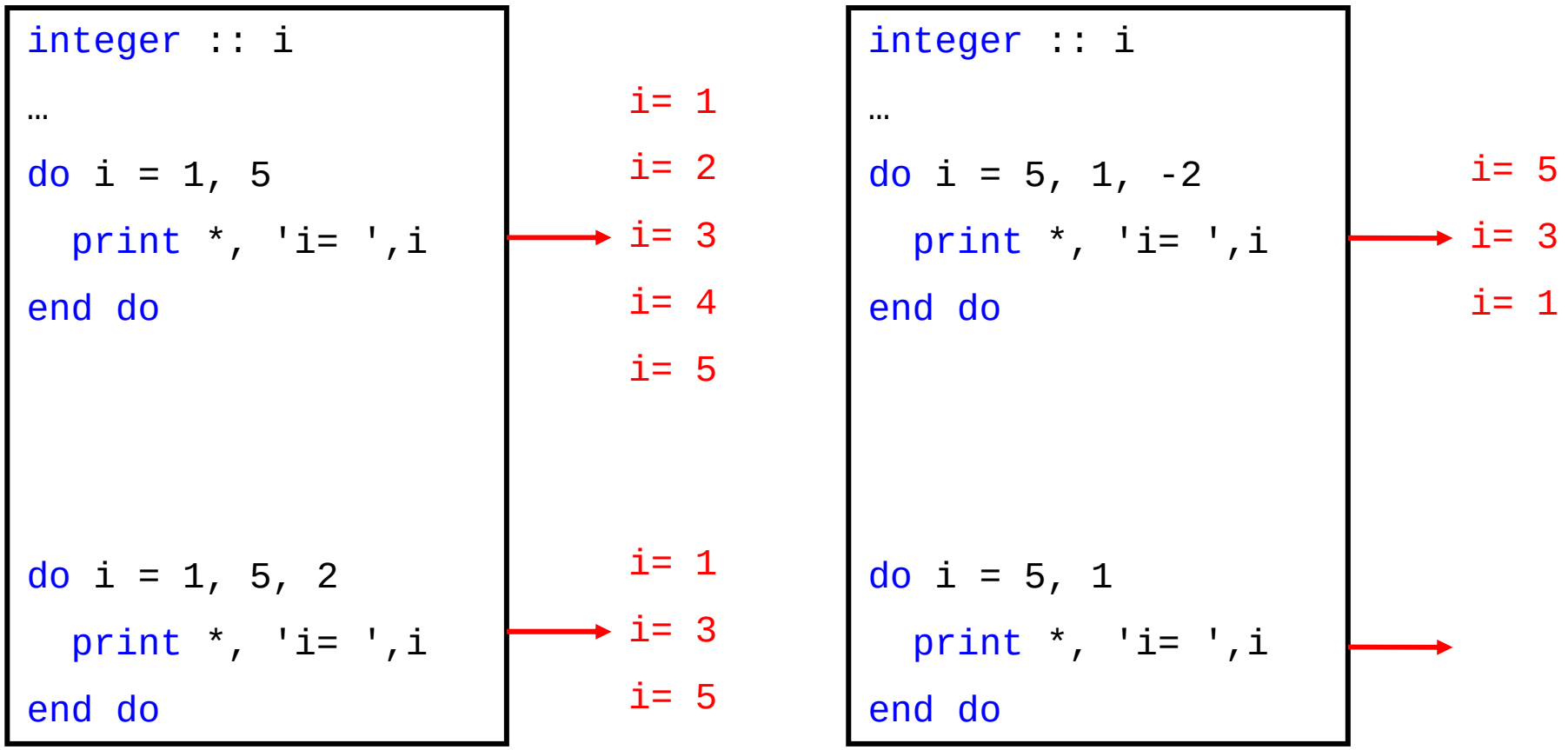
L'instruction DO : Boucle avec compteur (1)

```
[<nom> :] do <var> = <début>, <fin> [, <incrément>]
                    bloc
                end do [<nom>]
```

- var : variable de type INTEGER
- début, fin, incrément : expressions de type INTEGER
  - si incrément est omis, il est pris par défaut égal à 1.

- Remarques :
  - var doit être déclarée dans le programme même si elle n'apparaît pas dans bloc (variable de contrôle qui compte les tours)
  - var ne doit pas être modifiée dans le bloc régit par la boucle
  - Si fin < début, incrément est un entier négatif (sinon le programme n'entre pas dans la boucle)

L'instruction DO : Boucle avec compteur (2)



L'instruction DO : La boucle « tant que »

- Répétition conditionnelle (répétition dans laquelle la poursuite est régie par une condition)

```
[<nom> :] do while (<exp_log>)
                bloc
            end do [<nom>]
```

- exp_log : expression de type LOGICAL
- Remarques :
  - exp_log doit être définie avant de commencer l'exécution de la boucle
  - Arrêt de la boucle si la condition (testée au début de chaque itération) s'avère fausse.

L'instruction DO : boucle infinie

```
[<nom> :] do
                bloc
            end do [<nom>]
```

⇒ Besoin d'instructions qui modifient le déroulement de la boucle

L'instruction EXIT : sortie anticipée de boucle (1)

- L'instruction permet de sortir d'une boucle (n'importe quel type) et le programme reprend à la ligne qui suit le END DO.

```
do
  if (.not.<exp_log>) exit
  …
end do
```

⇔

```
do while (<exp_log>)
  …
end do
```

L'instruction EXIT : sortie anticipée de boucle (2)

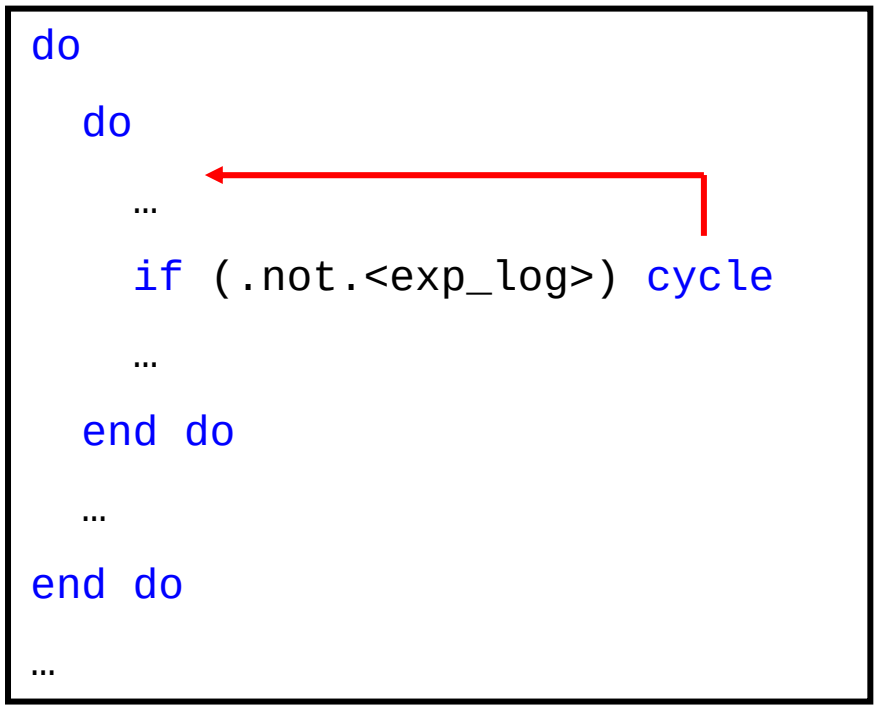
- Boucles imbriquées :
  - Sortie par défaut : boucle la plus intérieure
  - Sortie à un niveau supérieur : utiliser le nom de la boucle

```
do
  do
    if (.not.<exp_log>) exit
    …
  end do
  …
end do
…
```

```
bc1 do
      do
        if (.not.<exp_log>) exit bc1
        …
      end do
      …
    end do bc1
    …
```

L'instruction CYCLE : Bouclage anticipé

- L'instruction permet de rompre le déroulement d'une boucle (n'importe quel type) et le programme reprend à la ligne qui suit le DO.

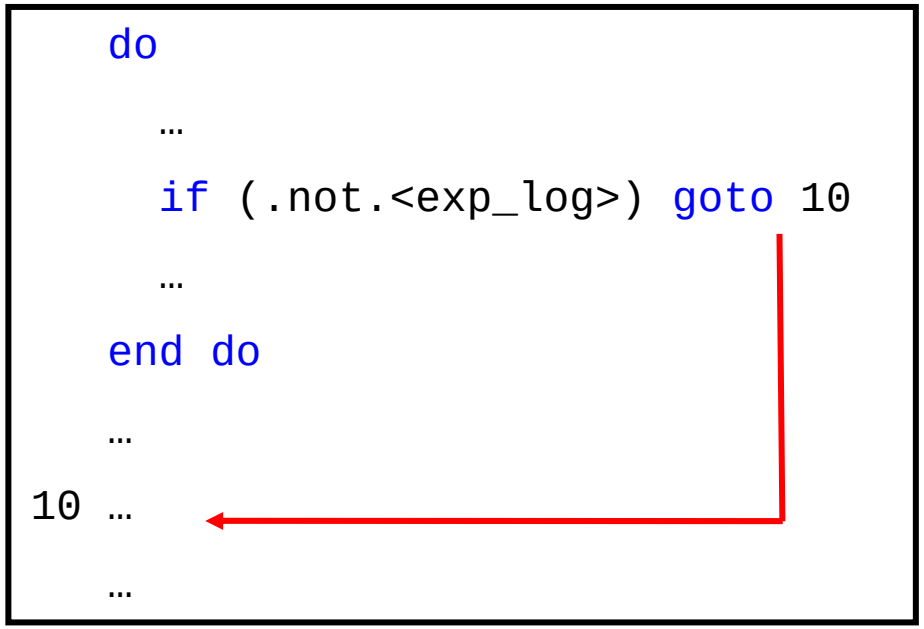


L'instruction GOTO : Redirection (1)

- L'instruction provoque un branchement à un emplacement quelconque d'un programme :

```
goto etiquette
```

- etiquette : nombre entier (sans signe) de 1 à 5 chiffres (non nul) qui doit figurer devant l'instruction à laquelle on souhaite se brancher



L'instruction GOTO : Redirection (2)

- A utiliser le moins souvent possible (lisibilité du programme)

```
! ***************************************************************************
      IMPLICIT DOUBLE PRECISION(A-H,O-Z)
      PI2 = 2*3.14159265358979323 8D0
      IF(L>=0) GOTO 1
      WRITE(*,*)' *** ARGUMENT L IN <YLM> MUST BE POSITIVE OR ZERO ***
      GOTO 12
    1 IF(L>400) GOTO 11
      MM = ABS(M)
      IF(MM>L) GOTO 9
! *** MONIC ASSOCIATED LEGENDRE POLYNOMIAL OF DEGREE L - IABS(M)
! *** AND NORMALIZATION
      PL = 1.0D0
      IF(MM==L) GOTO 3
      X = COS(TETA)
      PLM1 = 1.0D0
      PL = X
      A = 2.0D0*MM+1.0D0
      B = 1.0D0
      C = A
      PROD = 1.0D0/(A+2.0D0)
      U = PROD
      DO 2 I = MM+2,L
        PLM2 = PLM1
        PLM1 = PL
        PL = X*PLM1-U*PLM2
        A = A+1.0D0
        B = B+1.0D0
        C = C+2.0D0
        U = A/C*B/(C+2.0D0)
        PROD = PROD*U
    2 CONTINUE
      PL = PL/SQRT(PROD)
! *** NORMALIZATION OF THE WEIGHT FUNCTION SIN(TETA)**(2*IABS(M)+1)
    3 FAC = 0.5D0
      IF(MM==0) GOTO 5
      Y = SIN(TETA)
      Y = ABS(Y)
      IF(Y==0.0D0) GOTO 9
      TEST = 1.0D-78/Y
      A = 0.0D0
      DO 4 J = 1,MM
         A = A+2.0D0
         FAC = FAC*(1.0D0+1.0D0/A)
         IF(ABS(PL)<=TEST) GOTO 9
         PL = PL*Y
    4 CONTINUE
    5 PL = PL*SQRT(FAC/PI2)
! *** PARITY
      IF(M>0) THEN
         IF(MOD(M,2)==1) PL = -PL
      ENDIF
! *** MULTIPLY ASSOCIATED LEGENDRE FUNCTION
      IF(PHI==0.0D0) THEN
         YLMR = PL
         YLMI = 0.0D0
      ELSE
         ARG = M*PHI
         YLMR = PL*COS(ARG)
         YLMI = PL*SIN(ARG)
      ENDIF
      GOTO 10
    9 YLMR = 0.0D0
      YLMI = 0.0D0
   10 RETURN
   11 WRITE(*,*) ' *** L SO BIG THAT NORMALI
   12 WRITE(*,*) '      L =',L
      STOP
      END
```

L'instruction STOP : Arrêt du programme

- L'instruction permet l'arrêt de l'exécution du programme n'importe où (surtout ailleurs que end)

```
do
  do
    …
    if (.not.<exp_log>) then
      print *,'problème insurmontable'
      stop
    end if
    …
  end do
  …
end do
…
```

Chapitre 5. Les tableaux

Notions générales

- Les tableaux englobent les notions de
  - vecteur (tableau 1-D)
  - matrice (tableau 2-D)
  - grille (tableau 3-D et plus)

- On peut les générer de manière
  - statique (réservation de la mémoire au moment de la déclaration)
  - dynamique (réservation de la mémoire pendant l'exécution)

- Les tableaux sont traités comme
  - des listes de valeurs par certaines fonctions
  - des matrices au sens mathématique par d'autres

Déclaration statique d'un tableau

- Vecteur

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

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

- Tableau à 4 dimensions

```
integer,parameter :: n=3
real,dimension(1:n,1:n,-n:n,0:2*n) :: tab1
```

Remarques (1)

- Les *éléments* sont
  - utilisés comme n'importe qu'elle autre variable de son type (expression ou affectation)

```
n = t(3)*5 + 3

t(1) = 2*k + 5
```

  - de n'importe quel type (INTEGER, LOGICAL, REAL, COMPLEX, CHARACTER)

- Les *indices* prennent la forme de n'importe quelle **expression arithmétique de type INTEGER**.

```
t(3*p-1)
```

- Attention au débordement d'indices (pas forcément détecté à la compilation).

Remarques (2)

- Terminologie :
  - Le *rang* (rank) d'un tableau est le nombre de dimensions du tableau (maximum 7)
  - L'*étendue* (extend) d'un tableau dans une de ses dimensions est le nombre d'éléments dans cette dimension
  - La *taille* (size) est le produit des étendues dans chacune des dimensions
  - Le *profil* ou forme (shape) est la liste des étendues

```
real,dimension(-1:1,10,0:4) :: t
```

rang : 3

étendue : 3, 10, 5

taille : 150

profil : (3,10,5)

```
real,dimension(-1:5,0:9) :: tab1
real,dimension(7,2:11) :: tab2
```

Tableaux de même profil (7,10)

Ordre dans la mémoire

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

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

```
matrice(1,1)
matrice(2,1)
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)…
  end do
end do
```

inefficace

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

Construction de tableaux : *Array constructor*

- On peut définir le contenu d'un tableau à une dimension en utilisant un constructeur (*array constructor*) :

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

- Utilisation d'expressions dans un constructeur de tableau :

```
real, dimension(1:5) :: vecteur
real :: n,p
vecteur = (/ 3., n, n+p, 8.*p, p-n /)
```

- Utilisation de listes à boucle implicite

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

Construction de tableaux : la fonction RESHAPE (1)

- RESHAPE permet de fabriquer un tableau de profil donné, à partir d'un tableau à une dimension.
  - le premier argument contient la liste des valeurs à utiliser.
  - le second argument donne la forme de la matrice

```
integer, dimension(3,2) :: t
…
t = reshape((/ 1., 2., 3, 4., 5., 6. /), (/3,2/))
```

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

- Modification de l'ordre de remplissage

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

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

Remplissage où le 2nd argument varie en premier et le 1er argument varie en second

Construction de tableaux : la fonction RESHAPE (2)

- Remplissage par défaut (cas où le premier argument ne spécifie pas tous les éléments de la matrice) :

```
integer, dimension(3) :: t1 = (/ (i, i = 1,3) /)
integer, dimension(3,2) :: t2
…
t2 = reshape( t1, (/3,2/), order=(/2,1/), pad=(/0./))
```

$$t2 = \begin{pmatrix} 1. & 2. \\ 3. & 0. \\ 0. & 0. \end{pmatrix}$$

```
t2 = reshape( t1, (/3,2/), order=(/2,1/), pad=(/0., 1./))
```

$$t2 = \begin{pmatrix} 1. & 2. \\ 3. & 0. \\ 1. & 0. \end{pmatrix}$$

Opérations sur les tableaux : Affectation

- Affectation d'un seul élément

```
real, dimension(3,2) :: mat1
…
mat1(1,2) = 1.
```

- Affectation collective

```
real, dimension(1:3,1:2) :: mat1
real, dimension(-1:1,0:1) :: mat2
…
mat1 = 1.
mat1 = mat2(0,0)
mat1 = mat2  !possible si mat1 et mat2 sont de même profil
```

Opérations sur les tableaux : les expressions

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

- Opérateurs intrinsèques (+ - * / **) :
  - Le résultat correspond à un tableau dont chaque élément à la valeur et le type de l'opération entre éléments pris individuellement

```
real, dimension(1:3,1:3) :: mat1
real, dimension(0:2,-1:1) :: mat2
real, dimension(0:2) :: vec
…
mat1(:,1) = mat2(0,:) + vec
mat1 = mat1*mat2  !Ceci n'est pas un produit matriciel
```

- Fonctions élémentaires :
  - Opérations réalisées en prenant chaque élément individuellement

```
mat1 = exp(mat2)  !Ceci n'est pas l'exponentiel d'une matrice
```

Les sections : Sections régulières d'un vecteur

- Adressage de sections de tableaux en utilisant un indexage du type ([<début>] : [<fin>] [: <incrément>])

- Section continue

```
integer, dimension(10) :: v
integer, dimension(5) :: w
…
w = v(:5) + v(6:)          w = v(1:5) + v(3:7) + v(5:9)
```

- Section non continue

```
v(1:9:2) = w
v(10:2:-2) = w
```

Les sections : Sections de tableaux

- On peut étendre les notions vues précédemment à chaque dimension d'un tableau

```
integer, dimension(4,5) :: t
…
t(1:2,:3) = …
```

t(1,1) t(1,2) t(1,3)
t(2,1) t(2,2) t(2,3)

```
t(1:4:2,1:5)
```

t(1,1) t(1,2) t(1,3) t(1,4) t(1,5)
t(3,1) t(3,2) t(3,3) t(3,4) t(3,5)

```
t(:2,:2)   !Premier bloc (2x2) de t
t(:,1)     !Première colonne de t
t(::2,1)   !1 élément sur 2 de la première colonne de t
```

Les sections : Vecteurs d'indices

- Adressage des éléments d'un tableau en utilisant comme indice un vecteur d'entiers

```
integer, dimension(1:3) :: ind = (/ 1, 4, 7 /)
real, dimension(10,10) :: t
…
t(ind, 5) = …
```

t(1,5);t(4,5);t(7,5)

```
t(ind,ind)
```

t(1,1) t(1,4) t(1,7)
t(4,1) t(4,4) t(4,7)
t(7,1) t(7,4) t(7,7)

Les sections : Synthèse

<tableau(specif$_1$, specif$_2$, …, specif$_n$)>

- specif$_i$ est une des trois possibilités suivantes :
  - indice (expression entière)
  - indication de section régulière de la forme [<deb>] : [<fin>] [: <pas>]
  - vecteur d'indices (tableau d'entiers à une dimension)

Déclaration dynamique d'un tableau (1)

- Réservation/Libération d'une quantité d'espace mémoire au moment où c'est nécessaire

```
integer :: n
real,dimension(:),allocatable :: vec
real,dimension(:,:),allocatable ::  mat
…
n = 3
allocate(vec(1:n),mat(1:n,1:n))  !Allouer les tableaux

if (allocated(vec)) print *, 'vec alloué' !Tester allocation
if (allocated(mat)) print *, 'mat alloué'

deallocate(vec,mat)   ! Libérer la mémoire
```

- Gestion des erreurs

```
allocate(vec(1:n),stat=ierr)  !ierr déclaré comme INTEGER
if (ierr.ne.0) print *, 'Erreur'
```

  - Erreur si tableau déjà alloué ou mémoire insuffisante

Déclaration dynamique d'un tableau (2)

```
program exemple_allocate

 integer,parameter :: r=8
 integer :: ierr, n
 real(kind=r),dimension(:),allocatable :: vecteur

 n = 0
 ierr = 0

 do while (ierr.eq.0)
  n = n + 1024*1024
  if (allocated(vecteur)) deallocate(vecteur)
  allocate (vecteur(1:n),stat=ierr)
 end do
 n = n - 1024*1024

 print *, "Memoire disponible (MB) : ", n/(1024*1024)*r
 print *

end program exemple_allocate
```

```
"D:\Cours\Programmation_Bac2\Alex\Fortran\1-Programmes du cours\Chapitre03\De
 Memoire disponible (MB) :         1984

Press any key to continue
```

L'instruction WHERE

$$mat = \begin{pmatrix} 100. & 10. \\ 0.1 & 0. \end{pmatrix}$$

- Possibilité de soumettre une exécution d'affectation de tableau à une condition qui sera testée pour chaque élément du tableau (intérêt : conserver une écriture globale)

```
where(mat.gt.0.) mat = log10(mat)


where(mat.gt.0.)
     mat = log10(mat)
elsewhere
     mat = -100.
end where
```

$$mat = \begin{pmatrix} 2. & 1. \\ -1. & 0. \end{pmatrix}$$

$$mat = \begin{pmatrix} 2. & 1. \\ -1. & -100. \end{pmatrix}$$

Le masque conditionnant le WHERE et les tableaux concernés doivent avoir le même profil.

Opérations sur les tableaux : Fonctions intrinsèques (1)

- Remarques préliminaires :
  - Les fonctions peuvent parfois présenter deux arguments optionnels
    - *dim* : la fonction s'applique pour la dimension *dim*
    - *mask* : la fonction s'applique qu'aux éléments du tableau pour lesquels l'élément correspondant de *mask* a la valeur *vrai*.

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

- Fonctions « logiques »
  - all

```
all(<mask>)
```

    - fournit la valeur *vrai* si tous les éléments du tableau logique *mask* ont la valeur *vrai* (ou si mask est de taille zéro)
    - option : *dim*

```
if (all(mat.ge.0)) print *, 'Tous les éléments sont positifs'
all(mat.ge.5., dim = 1) ⟶ (/ F, F, T /)
all(mat.ge.5., dim = 2) ⟶ (/ F, F, F /)
```

Opérations sur les tableaux : Fonctions intrinsèques (2)

- Fonctions « logiques » (suite)
  - any

```
any(<mask>)
```

    - fournit la valeur *vrai* si au moins un des éléments du tableau logique *mask* a la valeur *vrai*
    - option : *dim*

  - count

```
count(<mask>)
```

    - fournit le nombre (entier) d'éléments du tableau logique *mask* a la valeur *vrai*
    - option : *dim*

```
count(mat.ge.0)
```

9

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

Opérations sur les tableaux : Fonctions intrinsèques (3)

- Fonctions de recherche d'extrema
  - maxval (minval)

```
maxval(<array>)
```

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

    - fournit la plus grande (petite) valeur du tableau *array*.
    - option : *dim, mask*

```
maxval(mat)                  → 9.
maxval(mat,dim = 1)          → (/ 3., 6., 9. /)
maxval(mat,dim = 2)          → (/ 7., 8., 9. /)
maxval(mat,mat.lt.5.)        → 4.
```

  - maxloc (minloc)

```
maxloc(<array>)
```

    - fournit les indices relatifs à la plus grande (petite) valeur de *array*.
    - option : *mask*

```
maxloc(mat)                  → (/ 3,3 /)
minloc(mat)                  → (/ 1,1 /)
```

Opérations sur les tableaux : Fonctions intrinsèques (4)

- Fonctions d'interrogation
  - lbound (ubound)

```
lbound(<array>)
```

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

    - Fournit la borne inférieure (supérieure) du tableau *array*.
    - option : *dim*

```
lbound(mat)          → (/ 1,1 /)
lbound(mat,dim = 1)  → 1
```

  - size

```
size(<array>)
```

    - Fournit la taille du tableau *array*. Si dim est présent, donne l'étendue
    - option : *dim*

```
size(mat)          → 9
size(mat,dim = 1)  → 3
```

  - shape

```
shape(<array>)
```

    - Fournit un tableau d'entiers de rang 1 correspondant au profil de *array*.

Opérations sur les tableaux : Fonctions intrinsèques (5)

- Opérations matricielles et vectorielles
  - dot_product (produit scalaire)

```
dot_product(<vec1>,<vec2>)
```

    - vec1, vec2 : tableaux de rang 1, de même taille et de types numériques ou logiques [any(vec1.and.vec2)]

  - matmul (produit matriciel)

```
matmul(<mat1>,<mat2>)
```

    - mat1, mat2 : tableaux de rang 1 ou 2 et de types numériques ou logiques
    - Respect des contraintes mathématiques habituelles du produit matriciel, càd en termes de profil (m,n)*(n,p) = (m,p), (m,n)*(n) = (m)

  - transpose (transposée)

```
transpose(<mat1>)
```

    - mat1 : tableaux de rang 2 et de type quelconque.

Opérations sur les tableaux : Fonctions intrinsèques (6)

- Fonctions sur les valeurs des éléments
  - product (sum)

```
product(<array>)
```

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

  - fournit le produit (la somme) des valeurs des éléments du tableau *array*.
  - résultat du même type que les éléments (INTEGER, REAL, COMPLEX)
  - option : *dim*

```
sum(mat)          → 45.
sum(mat,dim = 1)  → (/ 6., 15., 24. /)
sum(mat,dim = 2)  → (/ 12., 15., 18. /)
```

```
product(mat)          → 362880.
product(mat,dim = 1)  → (/ 6., 120., 504. /)
product(mat,dim = 2)  → (/ 28., 80., 162. /)
```

Opérations sur les tableaux : Fonctions intrinsèques (7)

- Fonctions de transformation
  - pack

```
pack(<array>, <mask> [, <vector>])
```

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

    - extrait certaines valeurs du tableau *array*.
    - vector décide de la longueur du résultat
    - résultat sous forme d'un tableau de rang 1

```
pack(mat,mat.ge.5.)          → (/ 5., 6., 7., 8., 9. /)
pack(mat,mat.ge.5.,(/1./))   → (/ 5. /)
pack(mat,mat.ge.5.,(/1., 1./)) → (/ 5., 6. /)
```

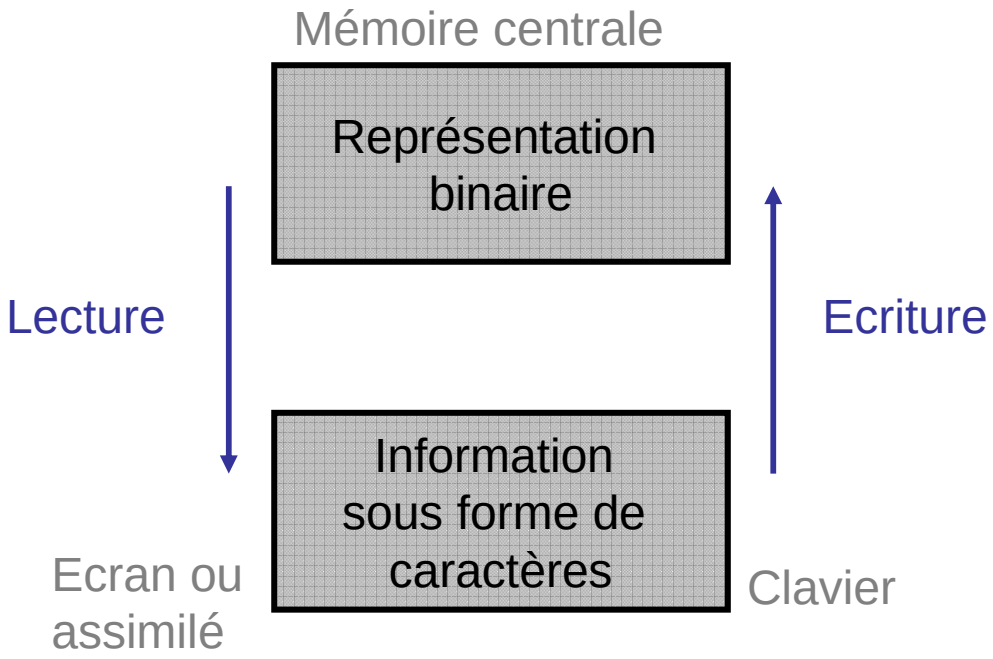
  - reshape
    - changement de forme (cfr précédemment)

Chapitre 6. Les opérations de lecture et d’écriture

Introduction

- Les ordres d'entrée/sortie permettent de transférer des informations entre la mémoire centrale et les unités périphériques (terminal, clavier, disque, imprimante,...)

- Les opérations d'entrée/sortie engendrent des conversions caractères alphanumériques/binaires



Syntaxe générale

- Lecture

```
read(périphérique, format [,options]) liste
read format , liste
```

- Ecriture

```
write(périphérique, format [,options]) liste
print format , liste
```

- Remarques :
  - les deuxièmes formes de syntaxe correspondent à l'entrée et la sortie dites standard (en général l'écran)
  - <options> reprend l'ensemble des arguments qui permettent une gestion des erreurs, un contrôle du changement de ligne,…

Entrées-sorties standards : Ecriture en format libre

```
print *, 'Energie totale = ', Etot, ' eV'
write(*,*) 'Energie totale = ', Etot, ' eV'
```

- Le * associé au format indique à l'instruction d'écriture de formater automatiquement les données. (Ex: données numériques écrites avec toutes leurs décimales représentées en machine)

- Le * associé au périphérique (write) correspond à la sortie standard.

Entrées-sorties standards : Lecture en format libre

```
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
  - ou des fins de ligne (à éviter)

- Les chaînes de caractères se notent entre apostrophes (') ou entre guillemets (")

Les formats : Introduction

- Motivations :
  - Lecture : adaptation à des données existantes non adaptées à un format libre
  - Ecriture : présentation des résultats

- Le format se présente comme une liste de **descripteurs** :

```
(i3,f5.3)
```

- Deux manières de donner le format d'entrée/sortie :
  - dans une instruction à part, dotée d'une étiquette

```
write(*,1000) liste

1000 format <liste_descripteur>
```

  - dans l'instruction d'entrée/sortie elle-même

```
write(*,'<liste_descripteur>') liste
```

Les formats : Descripteur des entiers

`Iw`

- w : nombre de caractères

- Lecture :
  - w caractères sont lus, y compris le signe éventuel et des espaces (interprétés comme zéro) devant.

```
read(*,'(i3,i4)') n,p
```

	n	p
^42^^23 →	42	23
^124547 →	12	4547
^-3^^-456 →	-3	-4

- Ecriture :
  - Impression de la valeur entière justifiée à droite dans les w caractères, avec le signe éventuel et des blancs pour compléter le champ w.

```
write(*,'(i3,i4)') n,p
```

n	p	
5	23	→ ^^5^^23
-122	45	→ ***^^45
-23	4521	→ -234521

Les formats : Descripteur des réels (sans exposant)

F*w.d*

- *w* : nombre de caractères (chiffres, point décimal, signe, blancs)
- *d* : nombre de chiffre décimaux

Valeur interprétée	Format	Ecriture
4.86	f4.1	^4.9
-5.23	f5.2	-5.23
4.25	f10.2	^^^^^^4.25

Les formats : Descripteur des réels (avec exposant)

E$w.d$

- w : nombre de caractères (chiffres, point décimal, signe, exposant, blancs)
- d : nombre de chiffre décimaux
- Attention : $w \geq d+6$

Valeur interprétée	Format	Ecriture
0.0123456	e13.6	^0.123456E-01
-47.678	e12.4	^-0.4768E+02
4.25	e12.3	^^^^0.425E01

- Remarques :
  - G$w.d$ $\Rightarrow$ choix automatique entre F$w.d$ et E$w.d$
  - Exposant à plus de deux chiffres : Ew.dEe avec e nombre de chiffres de l'exposant

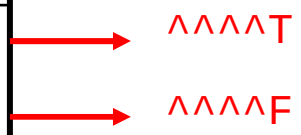
Les formats : Descripteur des logiques

L*w*

- *w*-1 blancs suivis de la lettre T pour une valeur .true. ou F pour une valeur .false.

```
write(*,'(l5)') test
```

test
.true.
.false.



Les formats : Descripteur pour caractères

Aw

- Si w > longueur de chaîne : blancs ajoutés en tête
- Si w < longueur de chaîne : seuls les w premiers caractères sont écrits
- Si w est absent : la longueur de chaîne fixe la longueur du champ de sortie

Chaîne	Format	Ecriture
'nom'	a	nom
'nom'	a5	^^nom
'nom'	a2	no

- Chaîne de caractères dans le format : reproduction telle quelle.

```
integer :: i=9
write(*,'('valeur de i=', i2)') i
```

→ Valeur^de^i=^9

Les formats : Descripteurs de mise en page

- Introduction des espaces :

*n*X

  - ignore (lecture) ou saute (écriture) les n caractères qui suivent

```
write(*,'(f6.2,3x,f5.3)') x,y
```

```
x = 2.5
y = 8.25
```

→ ^^2.50^^^8.250

- Positionnement dans le « tampon » :

T*c*

  - permet de se positionner au caractère de rang c

```
n = 5
p = 23
```

```
write(*,'(t10,'n=',i4,t20,'p=', i3) ') n,p
```

→ ^^^^^^^^^n=^^^5^^^^p=^23

```
write(*,'(t20,'n=',i4,t3,'p=', i3) ') n,p
```

→ ^^p=^23^^^^^^^^^^^n=^^^5

Les formats : Remarques

- Erreurs de descripteurs
  - Erreur non détectée par le compilateur (Ex: valeur entière décrite par le descripteur F)

- Ecriture sans retour à la ligne :

```
write(*,'(i4,i5,$)') n,p
write(*,'(i4,i5)',advance= 'no') n,p
```

- Descripteur de changement de ligne :

```
write(*,'(i4,/,i5)') n,p
```

- Répétition d'un descripteur :

```
write(*,'(2i4)') n,p
```

```
x = 132
y = 456
```

→ ^132^456

Les fichiers : Introduction

- Stockage des données (« enregistrements ») sur le disque.
- Généralement, on donne une extension .dat, .txt, .out, …
- Ils sont créés par des éditeurs de texte (Notepad, Emacs, Compaq Visual Studio,…)
- Fichier est dit
  - Formaté si les données sont stockées sous forme de caractères alphanumériques
  - Non formaté si les données sont stockées sous forme binaire
- Accessibilité :
  - Séquentiel : lecture de tous les enregistrements précédents
  - Direct : Ordre de lecture/écriture indique l'enregistrement à traiter.

Les fichiers : ouverture et création d'un fichier (1)

- Exploiter un fichier au sein d'un programme nécessite son ouverture. En Fortran, on utilise l'instruction open.

```
open(10,file='result.dat')                  !Forme compacte
open(10,file='result.dat',status='old') !Forme avec option
```

- L'instruction permet :
  - de connecter le nom de fichier à un **numéro d'unité logique** (numéro repris par la suite pour toute opération de lecture/écriture)
  - de spécifier le **mode** désiré : lecture, écriture ou les deux
  - d'indiquer le mode de **transfert** (avec ou sans formatage)
  - d'indiquer le mode d'**accès** (séquentiel ou direct)
  - de gérer les **erreurs** d'ouverture

Les fichiers : ouverture et création d'un fichier (2)

```
open([unit=]<num_unite> [,file=<nom_fichier>] [,status=<état>] &
[,access=<accès>] [,iostat=<result>] [,err=étiquette]          &
[,form=<format>] [,action=<action>] [,position=<pos>]          &
[,recl=<long_enreg>])
```

- num_unite :
  - numéro d'unité logique qui désigne le fichier dans la suite du programme (constante ou variable entière)
- nom_fichier :
  - chaîne ou variable de type CHARACTER spécifiant le nom du fichier
- état :
  - « UNKNOWN » (défaut), « OLD », « NEW », « SCRATCH » ou « REPLACE »
- accès :
  - « SEQUENTIAL » (défaut) ou « DIRECT »
- result :
  - variable entière qui contiendra le code erreur éventuel (valeur nulle si aucune erreur)

Les fichiers : ouverture et création d'un fichier (3)

```
open([unit=]<num_unite> [,file=<nom_fichier>] [,status=<état>] &
[,access=<accès>] [,iostat=<result>] [,err=étiquette]          &
[,form=<format>] [,action=<action>] [,position=<pos>]          &
[,recl=<long_enreg>])
```

- étiquette :
  - étiquette de renvoie à l'instruction à exécuter en cas d'erreur
- format :
  - « FORMATTED» (défaut) ou « UNFORMATTED » spécifiant le mode d'écriture, caractères ASCII ou binaire.
- action :
  - « READWRITE» (défaut), « READ » ou « WRITE »
- position : (uniquement pour fichier séquentiel)
  - « ASIS» (défaut – pas de modification de position si fichier déjà connecté , « REWIND » (début de fichier), « APPEND » (fin de fichier)
- long_enreg : (obligatoire si accès direct)
  - constante entière donnant la longueur en bytes de l'enregistrement

Les fichiers : Lecture (1)

- Syntaxe de base :

```
read([unit=]<num_unite>, [fmt=]<liste_descripteurs>) <liste>
```

```
read(10,'(2g15.6)') x,y
read(10,*) x,y
read(numfich,fmt=*) x,y,chaine
read(unit=20,fmt='(a)') chaine
```

Les fichiers : Lecture (2)

- Gestion des erreurs :
  - Le paramètre *iostat*

```
read(10,'(2g15.6)',iostat=ierr) x,y
```

    - ierr est un entier négatif : fin de fichier ou fin d'enregistrement
    - ierr est un entier positif : toute autre erreur (enregistrement insuffisant, fichier devenu inaccessible,…)
    - ierr est nul : aucune erreur

  - Les paramètres *end,err,eor*

```
read(10,'(2g15.6)',end=999,eor=888) x,y
```

Paramètre	Evènement
end	fin de fichier
err	erreur (autre que fin de fichier ou enregistrement)
eor	fin d'enregistrement

Les fichiers : Lecture (3)

```
program Exemple_fichier

   implicit none

   integer, parameter :: numfich=10
   character(len=12) :: nomfich
   integer :: ierr
   character(len=1) :: c


!Ouverture de fichier avec gestion d'erreur
   do
      write(*, '("nom du fichier : ")',advance='no')
      read(*,*) nomfich
      open(numfich,file=nomfich,status='old',iostat=ierr)
      if (ierr.eq.0) exit
      write(*,*) '-- fichier non trouve'
   end do

!Lecture du fichier et affichage à l'écran
   do
      read(numfich,'(a1)',advance='no', eor=88, end=99) c
      write(*,'(a1)',advance='no') c
      cycle
88    write(*,*)    !forcer un changement de ligne
      cycle
99    exit
   end do
   write(*,'(//"-- fin du fichier")')

end program Exemple_fichier
```

```
program Exemple_fichier2

   implicit none

   integer(kind=4),parameter                    :: lbuf=64000
   integer(kind=4)                              :: ierr
   character(len=512)                           :: line
   character(len=lbuf)                          :: buf

   open(unit=10,file='input.txt')
   buf = ' '
   do
      read(10,'(a)',iostat=ierr) line
      if (ierr.ne.0) exit
      if (len_trim(line).eq.0) cycle
      if (len_trim(buf) + len_trim(line).gt.lbuf) then
         write(*,*) 'Error: Buffer to small'
         stop
      end if
      buf = trim(adjustl(buf)) // trim(adjustl(line))
   end do

end program
```

Les fichiers : Ecriture

- Syntaxe de base

```
write([unit=]<num_unite>, [fmt=]<liste_descripteurs>) <liste>
```

```
write(10,‘(2g15.6)’) x,y
write(10,*) x,y
write(numfich,fmt=*) x,y,chaine
write(unit=20,fmt=‘(a)’) chaine
```

- Gestion des erreurs avec les paramètres *iostat* et *err* (voir lecture)

Les fichiers : Les opérations de positionnement

- Remonter au début du fichier :

```
rewind([unit=]<num_unite> [,iostat=<result>] [,err=étiquette])
```

- Remonter d'un enregistrement dans le fichier :

```
backspace([unit=]<num_unite> [,iostat=<result>] [,err=étiquette])
```

Les fichiers : lecture/écriture non formatée

```
open(10,file='result.dat',form='unformatted')
…
write(10,*) x,y
…
read(10,*) x,y
…
```

- Utilisation des lectures/écritures non formatées pour des fichiers destinés à être relus par l'ordinateur.

- Avantages :
  - Gain de place sur le disque
  - Gain de temps (pas besoin de conversion binaire ⇔ alphanumérique)
  - Maintien de la précision des variables numériques (sauf troncature)

- Désavantage :
  - Lecture/écriture sur un même type de machine

Les fichiers : Lecture/écriture en accès direct (1)

- Dès l'ouverture du fichier, on spécifie la longueur en octets de chaque enregistrement (par exemple, 4 pour un réel simple précision ou longueur de la chaîne) avec l'argument recl.

```
open(10,file='result.dat',access='direct',recl=4)
```

- Les *read* et *write* peuvent accéder à chaque enregistrement ***directement*** avec l'argument rec qui spécifie le numéro de l'enregistrement

```
write(10,rec=n) x
…
read(10,rec=n) x
```

Tous les enregistrements doivent être de même taille !

Les fichiers : Lecture/écriture en accès direct (2)

```
program Exemple_fichier3

   implicit none

   integer, parameter :: numfich=10, lge=48
   character(len=24) :: nomfich
   character(len=20) :: nom,prenom
   integer :: num,annee

   write(*,*) '--nom du fichier'
   read(*,*) nomfich

   open(numfich,file=nomfich,access='direct',recl=lge, &
        form='unformatted',status='new')

   write(*,*) 'numero enreg, prenom, nom, annee &
   &(numero nul pour finir)'

   do
      read(*,*) num,prenom,nom,annee
      if (num.eq.0) exit
      write(numfich,rec=num) prenom,nom,annee
   end do

end program Exemple_fichier3
```

```
program Exemple_fichier4

   implicit none

   integer, parameter :: numfich=10, lge=48
   character(len=24) :: nomfich
   character(len=20) :: nom,prenom
   integer :: num,annee

   write(*,*) '--nom du fichier'
   read(*,*) nomfich

   open(numfich,file=nomfich,access='direct',recl=lge, &
        form='unformatted',status='old')

   do
      write(*,'("numero enreg (numero nul pour finir) ")', &
            advance='no')
      read(*,*) num
      if (num.eq.0) exit
      read(numfich,rec=num) prenom,nom,annee
      write(*,'(1x,2a20,i5)') prenom,nom,annee
   end do

end program Exemple_fichier4
```

```
"D:\Cours\Programmation_Bac2\Programmes\Exemple_fichier\Exemple_fichier4\Debug\Exe...
 --nom du fichier
Nobelphys.dat
numero enreg (numero nul pour finir) 2
 pieter              zeeman              1902
numero enreg (numero nul pour finir) 3
 hendrik             lorentz             1902
numero enreg (numero nul pour finir) 9
 joseph              thomson             1906
numero enreg (numero nul pour finir) 0
Press any key to continue
```

Les fichiers : Les fichiers internes

- Fortran 90 offre la possibilité de coder ou décoder de l'information suivant un format. Ceci fait intervenir un tampon d'enregistrement (variable de type CHARACTER).

- Utilité :
  - Effectuer des conversions numérique ⇔ chaîne de caractères

Conversion en nombre

```
character(len=9) :: buf = '123456789'
integer :: n
real :: x
…
read(buf,'(i3,f6.2)') n,x
```

```
n=123, x=4567.89
```

Conversion en chaîne

```
character(len=80) :: tab
real :: x=3.54
…
write(tab,'("x : ",f8.3)') x
write(*,*) tab
```

```
x :    3.540
```

Les fichiers : L'instruction INQUIRE (1)

- L'instruction inquire permet de connaître les valeurs des arguments liés à un fichier ou une unité logique (formatage, accès, opérations autorisées).

- L'instruction possède des arguments ayant des noms identiques ou voisins à ceux de l'instruction *open*, avec la différence qu'elle en fournit la valeur.

```
inquire(file='input_field.txt',exist=exists)
if (.not. exists) then
   write(*,*) 'Error - Missing file - input_field.txt'
   stop
end if
open(10,file='input_field.txt')
```

Les fichiers : L'instruction INQUIRE (2)

- L'instruction inquire permet aussi d'obtenir la longueur de l'enregistrement que fournirait (en non formaté) une liste donnée :

```
inquire(iolength=long) x,(t(i), i=1,5),p
```


long contient la taille occupée par l'information correspondant à la liste

Les fichiers : fermeture

```
close([unit=]<num_unite> [,iostat=<result>] [,err=étiquette] &
       [,status=<état>])
```

- état :
  - « KEEP » (défaut – conserve le fichier – en conflit avec *scratch* de open) ou « DELETE » (efface le fichier, utile si le programme ne s'est pas déroulé comme prévu)

Chapitre 7. Les procédures

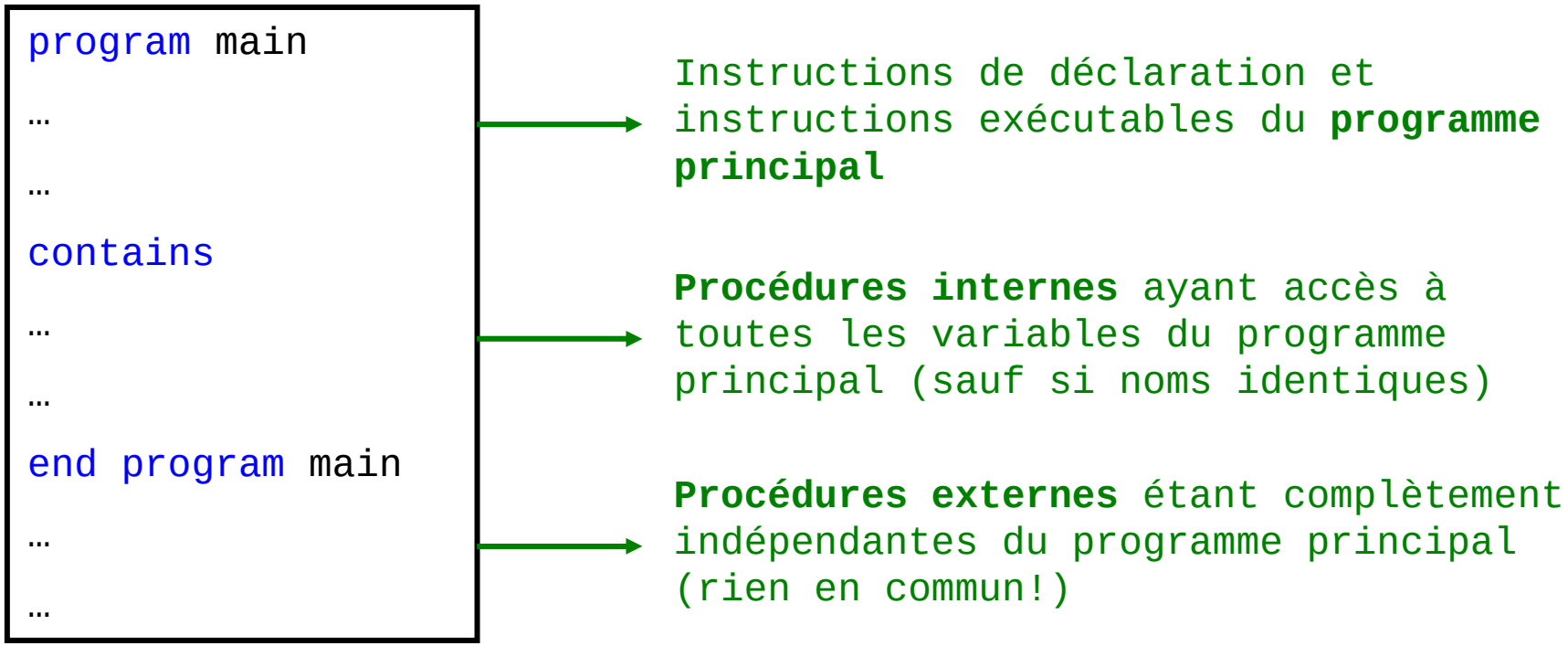
Introduction

- Fortran permet de scinder un programme en plusieurs parties :
  - Facilité de compréhension
  - Possibilité d'éviter des séquences d'instructions répétitives
  - Possibilité de partager des outils communs

- Deux sortes de procédures
  - Les fonctions
    - Entrée : un ou plusieurs arguments
    - Sortie : résultat unique (utilisable dans une expression)
  - Les sous-routines
    - Entrée : un ou plusieurs arguments
    - Sortie : un ou plusieurs arguments

Procédures internes ou externes

- Procédure dite
  - externe si séparée de toute autre unité de programme
  - interne si définie à l'intérieur d'une unité de programme
    - compilée en même temps que la procédure hôte
    - permet le partage d'informations (arguments + variables globales)



Variables locales ou globales (1)

- Les procédures internes ont accès à toutes les variables de la procédure hôte, …

```
program main
integer :: i=5
call Sousroutine
stop
contains
  subroutine Sousroutine
     integer :: j=3
     print *,i,j
   end subroutine Sousroutine
end program main
```

```
5 3
```

i est une variable du programme principal. Les procédures internes y ont accès => **Variable globale**

j est une variable de la procédure interne. Le programme principal n'y a pas accès => **Variable locale**

Variables locales ou globales (2)

- …, sauf celles qui sont déclarées avec un nom identique dans la procédure interne. La **variable globale est masquée**!

```
program main
integer :: i=5
call Sousroutine
stop
contains
  subroutine Sousroutine
     integer :: i=3
     print *,i
   end subroutine Sousroutine
end program main
```

3

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

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

C'est le « i local » qui est imprimé

Les sous-routines : Syntaxe générale

- Structure d'une sous-routine :

```
subroutine <nom> [(<dummy_arg1> [,<dummy_arg2>,…,<dummy_argn>])]
  : Instructions de déclaration des dummy arguments
  : Instructions de déclaration des variables locales

  : Instructions exécutables

end subroutine <nom>
```

- Appel d'une sous-routine :

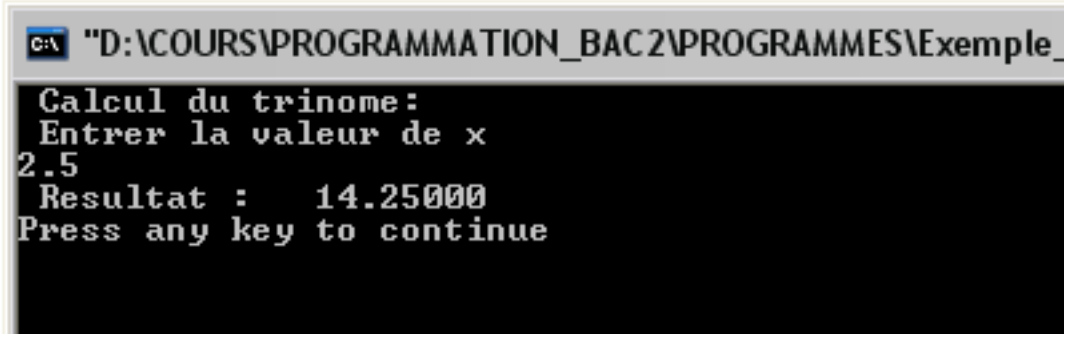
```
call <nom> [(<actual_arg1> [,<actual_arg2>,…,<actual_argn>])]
```

- Il y a autant de <actual_arg> que de <dummy_arg>. Dans le cas de variables, les arguments sont de mêmes type et kind.
- Les dummy_arg peuvent être des variables, des chaînes, des tableaux, des pointeurs, des noms de procédures
- Il est possible de sortir d'une sous-routine grâce à l'instruction return

MCours.com

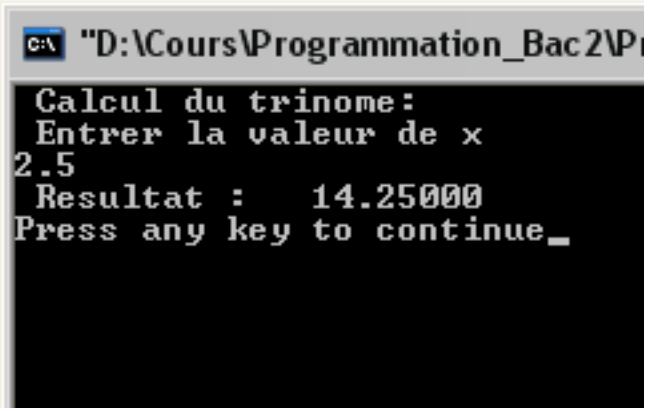
Les sous-routines : Exemple (sous-routine interne)

```
program Exemple_sousroutine

   implicit none

   real :: a=1.,b=2.,c=3.
   real :: val
   real :: res

   print *,'Calcul du trinome:'
   print *,'Entrer la valeur de x'
   read *,val

   call trinome(val,res)

   print *,'Resultat :', res

   contains
      subroutine trinome(x,y)
         real :: x,y

         y = a*x**2+b*x+c     !a,b,c : variables globales

      end subroutine trinome

end program Exemple_sousroutine
```



Les sous-routines : Exemple (sous-routine externe)

```
program Exemple_sousroutine2

   implicit none

   real :: a=1.,b=2.,c=3.
   real :: val
   real :: res

   print *,'Calcul du trinome:'
   print *,'Entrer la valeur de x'
   read *,val

   call trinome(val,a,b,c,res)

   print *,'Resultat :', res

end program Exemple_sousroutine2

subroutine trinome(x,a,b,c,y)

   implicit none

   real :: x,y
   real :: a,b,c

   y = a*x**2+b*x+c

end subroutine trinome
```



Les sous-routines : Utilisation avancée du CALL

- Arguments positionnels :
  - Mettre simplement les actual arguments dans le même ordre que les dummy arguments

```
call trinome(3.,a,b,c,res)
```

- Arguments à mot clé :
  - Rappeler le nom des dummy arguments dans l'instruction *call*
  - Avantage : on peut modifier l'ordre des arguments

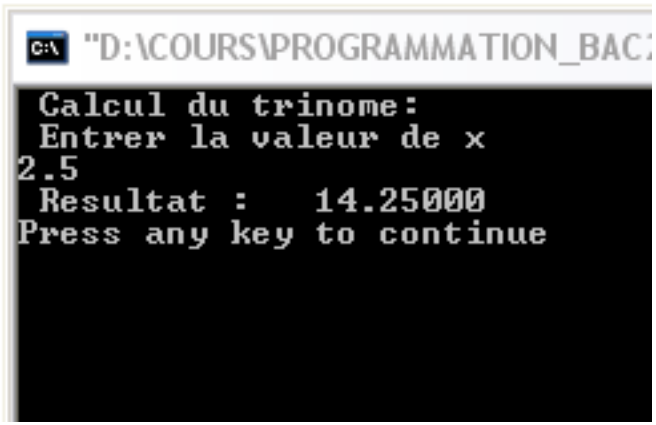
```
call trinome(x=3., a=a, b=b, c=c, y=res)
call trinome(y=res, a=a1, b=a2, c=a3, x=3.)
```

- Remarque :
  - On peut mélanger les deux méthodes. Le principe est que dès qu'un argument est donné avec son keyword, les arguments suivants doivent l'être également

Les fonctions : Introduction

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

```
program Exemple_fonction

   implicit none

   real :: a1=1.,a2=2.,a3=3.
   real :: val
   real :: trinome

   print *,'Calcul du trinome:'
   print *,'Entrer la valeur de x'
   read *,val

   print *,'Resultat :',  trinome(val,a1,a2,a3)

end program Exemple_fonction

function trinome(x,a,b,c)

   implicit none

   real :: trinome
   real :: x,a,b,c

   trinome = a*x**2+b*x+c

end function trinome
```



Les fonctions : Remarques

- Les dummy arguments d'une fonction doivent être déclarés comme ceux d'une sous-routine.
- Il est nécessaire de spécifier le type et le kind d'une fonction.
- Une fonction peut être interne ou externe
- Il est possible de sortir d'une fonction à tout moment grâce à l'instruction return

Les différentes sortes d'argument

- Les dummy arguments qui sont des variables ou des tableaux peuvent être déclarés avec l'attribut intent :
  - INTENT(IN) : l'argument est un *argument d'entrée* dont la valeur est utilisée par la procédure mais dont la valeur ne doit pas être modifiée.
  - INTENT(OUT) : l'argument est un *argument de sortie* dont la procédure doit attribuer une valeur mais ne jamais l'utiliser.
  - INTENT(INOUT) : l'argument est à la fois un *argument d'entrée et un argument de sortie*.

```
subroutine test(x,y,z)
real,intent(in) :: x
real,intent(out) :: y
real,intent(inout) :: z
y = 2.*x
z = z+y
end subroutine test
```

L'utilisation de l'attribut INTENT est vivement conseillé pour fiabiliser les codes

Les interfaces : Motivation

- Fiabiliser les appels de procédures

```
call optimist(5.25)
…
end program


subroutine test(n)
integer,intent(in) :: n
…
```

Aucun diagnostic à la compilation.
Lors de l'exécution, la sous-routine recevra le « motif binaire » de 5.25 en type réel et il l'interprétera comme un entier

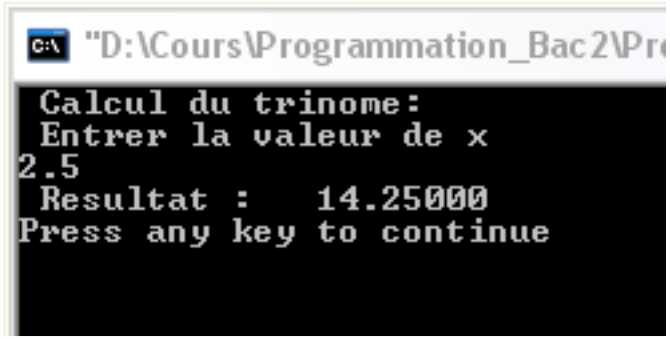
=> Erreur assurée!

- Transmettre des tableaux de taille quelconque
- Transmettre des procédures
- ...

A découvrir ultérieurement!

Les interfaces : exemple

```
program Exemple_sousroutine3

   implicit none

   real :: a=1.,b=2.,c=3.
   real :: val
   real :: res

   interface
      subroutine trinome(x,a,b,c,y)
         real,intent(in) :: x,a,b,c
         real,intent(out) :: y
      end subroutine
   end interface

   print *,'Calcul du trinome:'
   print *,'Entrer la valeur de x'
   read *,val

   call trinome(val,a,b,c,res)

   print *,'Resultat :', res

end program Exemple_sousroutine3

subroutine trinome(x,a,b,c,y)

   implicit none

   real,intent(in) :: x,a,b,c
   real,intent(out) :: y

   y = a*x**2+b*x+c

end subroutine trinome
```



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

L'attribut SAVE (1)

- 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 :: n
```

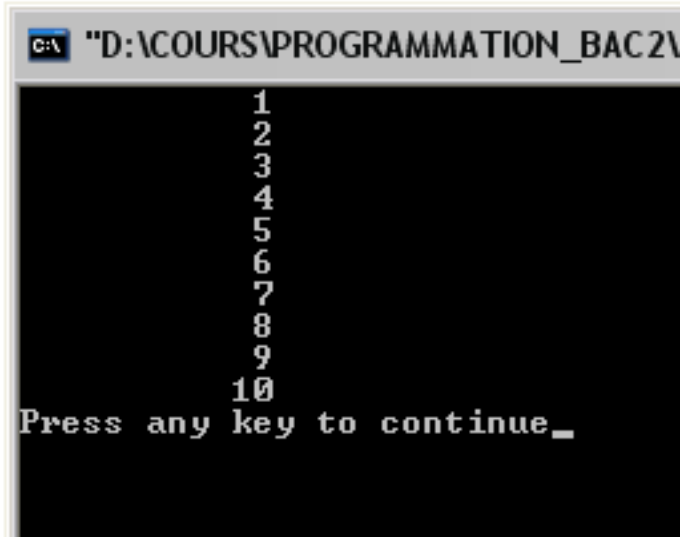
- On peut par ailleurs, au moment de la spécification de cette variable locale, lui donner une valeur initiale.

```
integer,save :: n = 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.

L'attribut SAVE (2)

```
program Exemple_save

 integer :: n

 interface
  subroutine Sousroutine
  end subroutine Sousroutine
 end interface

 do n = 1, 10
  call Sousroutine
 end do

end program Exemple_save

subroutine Sousroutine

 integer,save :: i = 0

 i = i + 1

 print *, i

end subroutine Sousroutine
```



Procédures internes ou externes ?

- Avantages d'une procédure interne :
  - Pas nécessaire de construire une interface.
  - La procédure interne a accès à toutes les variables de la procédure hôte (sauf si utilisation de noms identiques). Il n'est donc pas nécessaire de passer les variables globales en argument.

- Désavantges d'une procédure interne :
  - Risque important de modifier accidentellement les variables de la procédure hôte (variables globales masquées).
  - La procédure interne n'est accessible que par la procédure hôte qui la contient. Pour une utilisation par plusieurs procédures, il faut absolument créer des procédures externes.

Il est recommandé de construire a priori des procédures externes, sauf s'il y a un avantage indéniable à faire autrement.

Chapitre 8. Les modules

Notions générales

- Un module est une unité de programme indépendante qui peut contenir différents éléments :
  - des déclarations sur la précision utilisée (simple ou double précision)
  - des instructions de déclaration (constantes ou variables partagées)
  - des interfaces de procédures
  - des procédures

```
module <Nom du module>
  :
  : Instructions de déclaration
  :
contains
  :
  : Instructions exécutables
  :
end module <Nom du module>
```

```
program <Nom du programme>
  use <Nom du module>
  :
  : Instructions de déclaration
  :


  :
  : Instructions exécutables
  :
end program <Nom du programme>
```

Module pour la précision utilisée

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

```
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
real(kind=r) :: z = 1.02548_r
…
end program main
```

Module de constantes : Utilisation

- Un module permet de définir des constantes, des variables ayant l'attribut *parameter*.

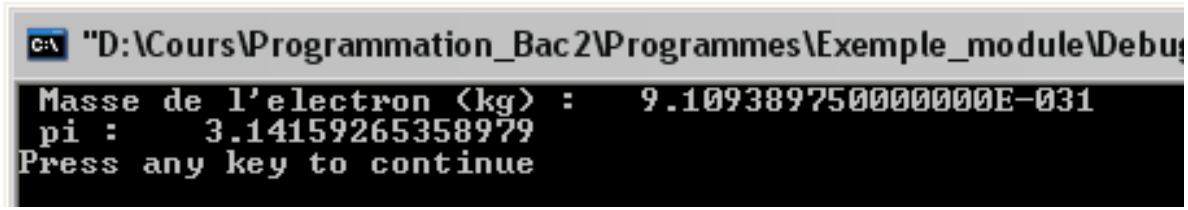
```
module const_fond
real(kind=8),parameter :: em   = 9.10953d-31,  &
                          ec   = 1.602189d-19, &
                          hbar = 1.054589d-34, &
                          pi   = 3.141592654d0
end module const_fond
```

- Utilisation
  - Tout le module : use const_fond
  - Certaines constantes : use const_fond, only: em,ec
  - Certaines constantes renommées : use const_fond, only: masse => em

Le **em** du module const_fond est alors connu sous le nom **masse** dans la procédure qui utilise le module.

Module de constantes : Exemple

```
module parameters

   real(kind=8),parameter                    :: c=299792458.0d0
   real(kind=8),parameter                    :: pi=3.14159265358979d0
   real(kind=8),parameter                    :: eps0=8.85418782d-12
   real(kind=8),parameter                    :: mu0=1.25663706143d-6
   real(kind=8),parameter                    :: ec = 1.60217733d-19
   real(kind=8),parameter                    :: em = 9.10938975d-31
   complex(kind=8),parameter                 :: ci=(0.0d0,1.0d0)

end module parameters


program Exemple_module

   use parameters, only: masse => em,pi

   print *,"Masse de l'electron (kg) : ", masse
   print *,"pi : ", pi

end program Exemple_module
```



Module de données partagées : Utilisation

- Un module peut contenir des données (variables, tableaux,...) qui seront utilisées par plusieurs procédures. Il est *vivement recommandé* de spécifier l'attribut *save*.

```
module datas
real(kind=8),save :: rayon,hauteur
end module datas
```

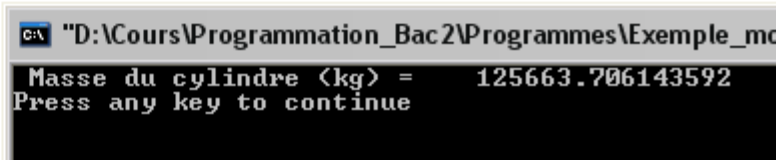
- Utilisation
  - Tout le module : use datas
  - Certaines données : use datas, only: rayon

- L'argument *save* assure que les données préservent leur valeur lors du passage à travers différentes procédures.

Module de données partagées : Exemple

```
module kinds
   integer(kind=2),parameter :: r=8
end module

module const_fond
   use kinds
   real(kind=r),parameter :: em   = 9.10653E-31_r
   real(kind=r),parameter :: ec   = 1.602189E-19_r
   real(kind=r),parameter :: hbar = 1.054589E-34_r
   real(kind=r),parameter :: pi   = 3.14159265358979_r
end module const_fond

module datas
   use kinds
   real(kind=r),save :: rayon, hauteur
end module datas

program module_donnees

   use kinds
   use datas
   implicit none

   interface
      function masse(rho)
      use kinds
      real(kind=r) :: masse
      real(kind=r),intent(in) :: rho
      end function masse
   end interface

   real(kind=r) :: rho = 1000.0_r

   rayon = 2.0
   hauteur = 10.0

   print *, 'Masse du cylindre (kg) = ', masse(rho)

end program module_donnees
```

```
function masse(rho)

   use kinds
   use const_fond
   use datas

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

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

end function masse
```



Module avec des interfaces : Utilisation

- Un module peut rassembler les interfaces de toutes les procédures rencontrées dans un projet. Il suffit de construire un seul exemplaire de ces interfaces, même si les procédures sont utilisées à plusieurs reprises dans le projet.

```
module list_int
  interface
  function masse(rho)
  use kinds
  real(kind=r):: masse
  real(kind=r),intent(in) :: rho
  end function masse
  …
  end interface
end module list_int
```

- Utilisation :
  - Tout le module : use list_int
  - Certaines procédures (pref.) :
    use list_int, only: masse

Module avec des interfaces : Exemple

```
module kinds
   integer(kind=2),parameter :: r=8
end module

module const_fond
   use kinds
   real(kind=r),parameter :: em   = 9.10653E-31_r
   real(kind=r),parameter :: ec   = 1.602189E-19_r
   real(kind=r),parameter :: hbar = 1.054589E-34_r
   real(kind=r),parameter :: pi   = 3.14159265358979_r
end module const_fond

module datas
   use kinds
   real(kind=r),save :: rayon, hauteur
end module datas

module list_interface
   interface
      function masse(rho)
      use kinds
      real(kind=r) :: masse
      real(kind=r),intent(in) :: rho
      end function masse
   end interface
end module list_interface

program module_interface

   use kinds
   use datas
   use list_interface
   implicit none

   real(kind=r) :: rho = 1000.0_r

   rayon = 2.0
   hauteur = 10.0

   print *, 'Masse du cylindre (kg) = ', masse(rho)

end program module_interface
```

```
function masse(rho)

   use kinds
   use const_fond
   use datas

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

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

end function masse
```

```
"D:\Cours\Programmation_Bac2\Programmes\Exemple_mo
 Masse du cylindre (kg) =    125663.706143592
Press any key to continue_
```

Module de procédures : Utilisation

- Un module peut encapsuler des procédures. Il est, dès lors, inutile de créer des interfaces.

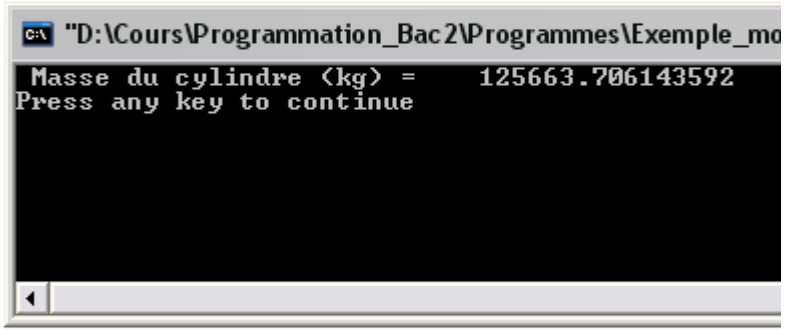
```
module procedures


contains
  function masse(rho)
  use kinds
  real(kind=r):: masse
  real(kind=r),intent(in) :: rho
  end function masse
  …
end module procedures
```

1. Il est possible de mettre plusieurs procédures.
2. Un module de procédures peut contenir des procédures internes.

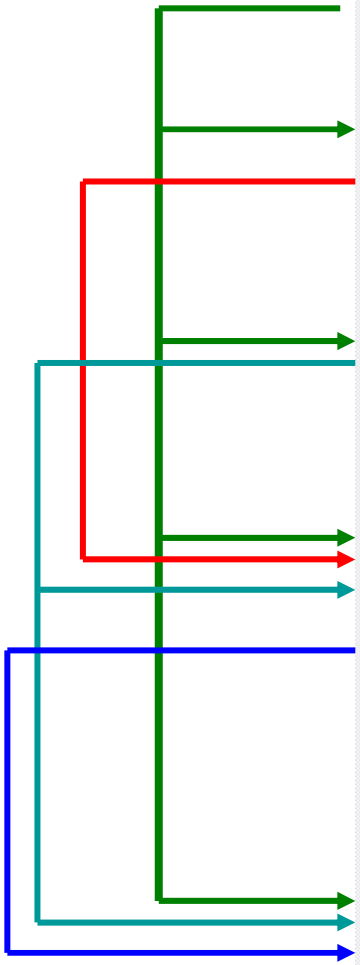
- Utilisation :
  - Toutes les procédures : use procedures
  - Certaines procédures : use procedures, only: masse

Module de procédures : Exemple

```
module kinds
   integer(kind=2),parameter :: r=8
end module

module const_fond
   use kinds
   real(kind=r),parameter :: em   = 9.10653E-31_r
   real(kind=r),parameter :: ec   = 1.602189E-19_r
   real(kind=r),parameter :: hbar = 1.054589E-34_r
   real(kind=r),parameter :: pi   = 3.14159265358979_r
end module const_fond

module datas
   use kinds
   real(kind=r),save :: rayon, hauteur
end module datas

module procedures

   contains
   function masse(rho)
      use kinds
      use const_fond
      use datas

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

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

   end function masse

end module procedures

program module_procedure
   use kinds
   use datas
   use procedures
   implicit none

   real(kind=r) :: rho = 1000.0_r

   rayon = 2.0
   hauteur = 10.0

   print *, 'Masse du cylindre (kg) = ', masse(rho)

end program module_procedure
```



Dépendance entre modules



```
module kinds
   integer(kind=2),parameter :: r=8
end module

module const_fond
   use kinds
   real(kind=r),parameter :: em   = 9.10653E-31_r
   real(kind=r),parameter :: ec   = 1.602189E-19_r
   real(kind=r),parameter :: hbar = 1.054589E-34_r
   real(kind=r),parameter :: pi   = 3.14159265358979_r
end module const_fond

module datas
   use kinds
   real(kind=r),save :: rayon, hauteur
end module datas

module procedures

   contains
   function masse(rho)
      use kinds
      use const_fond
      use datas

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

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

   end function masse

end module procedures

program module_procedure
   use kinds
   use datas
   use procedures
   implicit none

   real(kind=r) :: rho = 1000.0_r

   rayon = 2.0
   hauteur = 10.0

   print *, 'Masse du cylindre (kg) = ', masse(rho)

end program module_procedure
```

Dépendance interdite :

Un module qui s'appelle lui-même directement ou indirectement

Chapitre 9. Les structures (ou types dérivés)

Motivation

- La structure (*type dérivé*) permet de désigner sous **un seul nom** un ensemble de valeurs pouvant être de **types différents**.

- L'accès à chaque élément de la structure (*champ*) se fera, non plus par indication de position, mais par son nom au sein de la structure.

Déclaration d'une structure

- Pour définir des variables de type structure, il faut procéder en deux étapes :
  - Définir à l'aide des déclarations appropriées, les types des différents champs:

```
type atom
  character(len=2) :: symbol
  integer :: Z
  real :: A
end type atom
```

  - Déclarer (« classiquement ») une ou plusieurs variables du nouveau type ainsi créé

```
type(atom) :: carbon, silicium, germanium, tin, lead
```

- Remarque :
  - Possible de déclarer des constantes de type structures

Utilisation de structures

- Utilisation par champs :
  - Chaque champs d'une structure peut être manipulé comme n'importe quelle variable du type correspondant (real, complex, character,…).

```
carbon%symbol = 'C'
print *, carbon%Z
read *, carbon%A
silicium%Z = carbon%Z + 8
```

- Utilisation globale :

```
carbon = atom('C',6,12.0107)
print *, carbon
read *, silicium
```

Imbrication de structures (1/3)

- Structure comportant des tableaux ou des chaînes de caractères

```
type pers
  character(len=16) :: nom
  integer,dimension(3) :: qtes
end type pers
type(pers) :: employe = pers('Durand', (/2,5,3/))
…
print *,employe%nom(1:3)   ⟶ Dur
print *,employe%qtes(2)    ⟶ 5
```

Imbrication de structures (2/3)

- Structure comportant une autre structure

```
type point
   real :: x_coord, y_coord
end type point
type cercle
   real :: rayon
   type(point) :: centre
end type cercle

type(cercle) :: rond
…
rond%centre%x_coord = 5.0
```

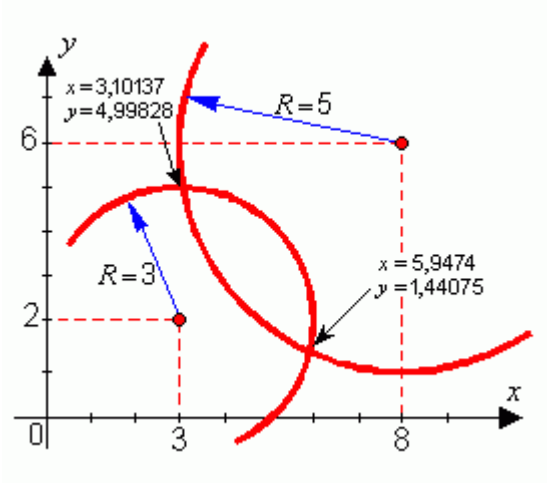
Imbrication de structures (3/3)

- Tableaux de structures

```
type atom
  character(len=2) :: symbol
  integer :: Z
  real :: A
end type atom
type(atom),dimension(1:118) :: PeriodicTable
…
PeriodicTable(1) = atom('H',1,1.0079)
PeriodicTable(8)%symbol = 'O'
PeriodicTable(8)%Z = 8
PeriodicTable(8)%A = 15.999
print *,PeriodicTable(1)%symbol
read *,PeriodicTable(6)
```

Exemple

```
"D:\COURS\PROGRAMMATION_BAC2\PROG
 Intersection
   3.101371        4.998286
   5.947410        1.440737
Press any key to continue_
```




```
program Exemple_type

   implicit none

   type point
      real :: x_coord,y_coord
   end type point
   type circle
      real :: radius
      type(point) :: centre
   end type circle
   type(circle) :: circ1,circ2
   type(point) :: point_p,point_q
   real,parameter :: pi = 3.14159265
   real :: a,b,c
   real :: buf
   real :: delta,deltaxp,deltaxq

   circ1%centre%x_coord = 8.0
   circ1%centre%y_coord = 6.0
   circ1%radius = 5.0

   circ2%centre%x_coord = 3.0
   circ2%centre%y_coord = 2.0
   circ2%radius = 3.0

   a = 2.*(circ2%centre%x_coord-circ1%centre%x_coord)
   b = 2.*(circ2%centre%y_coord-circ1%centre%y_coord)
   c = (a/2.)**2+(b/2.)**2-circ2%radius**2+circ1%radius**2
   buf = (a**2+b**2)
   delta = (2.*a*c)**2-4.*buf*(c**2-b**2*circ1%radius**2)
   point_p%x_coord = circ1%centre%x_coord+(2.*a*c-sqrt(delta))/(2.*buf)
   point_q%x_coord = circ1%centre%x_coord+(2.*a*c+sqrt(delta))/(2.*buf)
   deltaxp = (point_p%x_coord-circ1%centre%x_coord)
   deltaxq = (point_q%x_coord-circ1%centre%x_coord)

   if (b.ne.0.) then
      point_p%y_coord = circ1%centre%y_coord+(c-a*(deltaxp))/b
      point_q%y_coord = circ1%centre%y_coord+(c-a*(deltaxq))/b
   else
      point_p%y_coord = circ1%centre%y_coord+b/2.+ &
                        sqrt(circ2%radius**2-((2*c-a**2)/2.*a)**2)
      point_q%y_coord = circ1%centre%y_coord-b/2.+ &
                        sqrt(circ2%radius**2-((2*c-a**2)/2.*a)**2)
   end if

   print *,'Intersection'
   print *, point_p
   print *, point_q

end program Exemple_type
```

Chapitre 10. La librairie graphique DFLIB

Introduction

- La librairie DFLIB permet de réaliser des graphiques avec Fortran 90.

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

- L'appel à la librairie se fait comme un module :

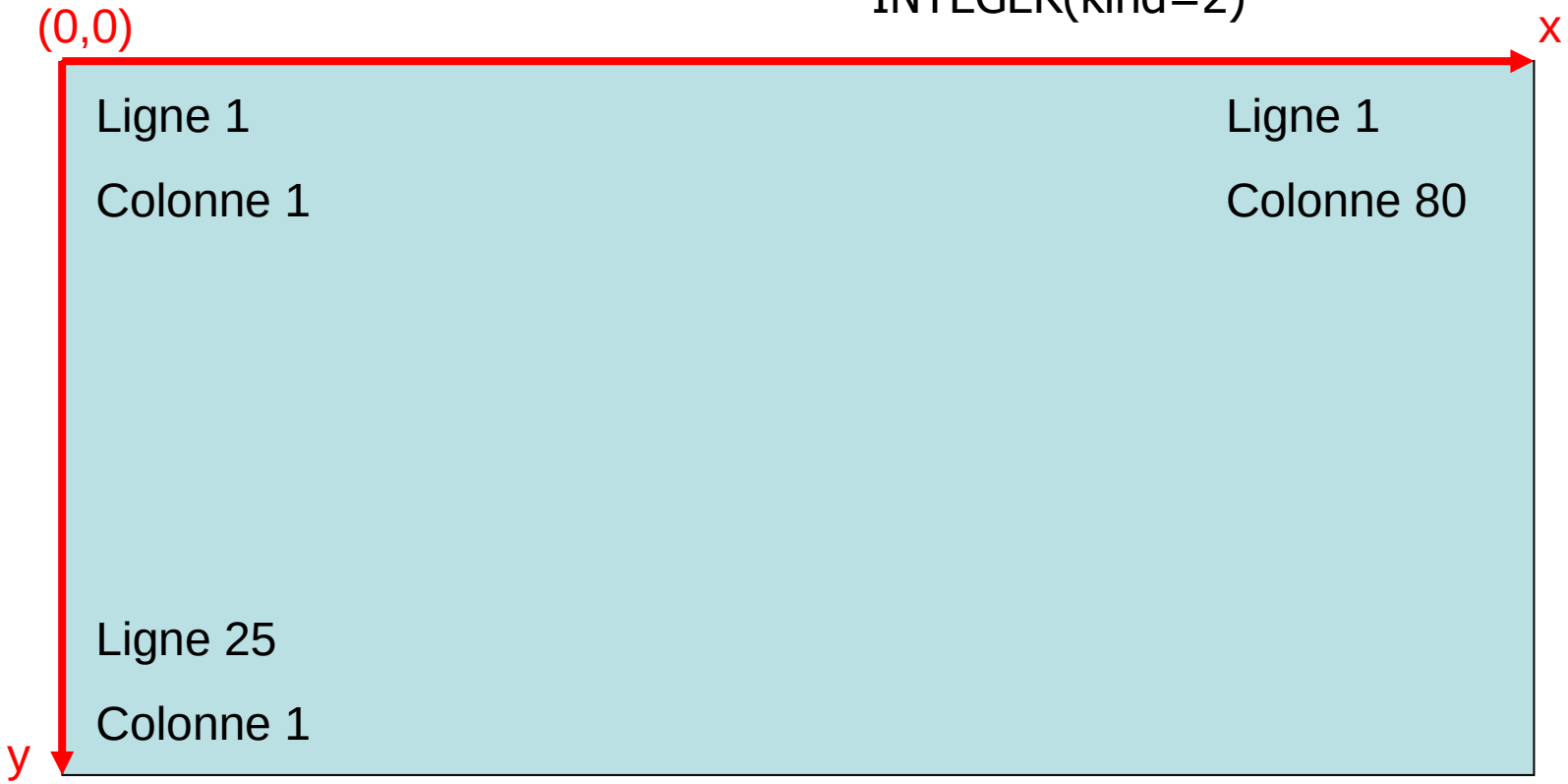
```
use dflib
```

Comprendre les systèmes de coordonnées :

- Coordonnées du Texte
- Coordonnées graphiques
  - Coordonnées physiques
  - Coordonnées Viewport
  - Coordonnées Window

Système de coordonnées : Texte et physique

- Les coordonnées du texte :
  - Lignes : de 1 à 25
  - Colonnes : de 1 à 80
  - Position : (ligne , colonne)
- Les coordonnées physiques :
  - x : 0 à 639 pixels
  - y : 0 à 479 pixels
  - Position : (x,y) où x,y sont des INTEGER(kind=2)



Rq : Valeurs données pour un écran 640x480

Système de coordonnées : Changement d'origine

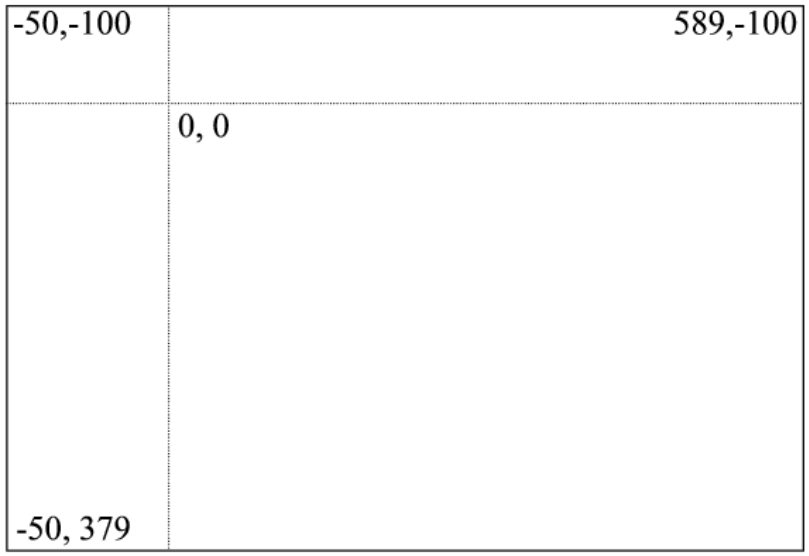
- On peut modifier l'origine du sytème de coordonnées avec la sous-routine setvieworg, dont les deux premiers arguments sont des INTEGER(kind=2) :

```
call setvieworg(50_2,100_2,xy)
```

- *xy* contient en sortie les coordonnées physiques de l'ancienne origine. A la première utilisation, ce serait (0,0). *xy* doit être déclaré comme

```
type(xycoord) :: xy
```

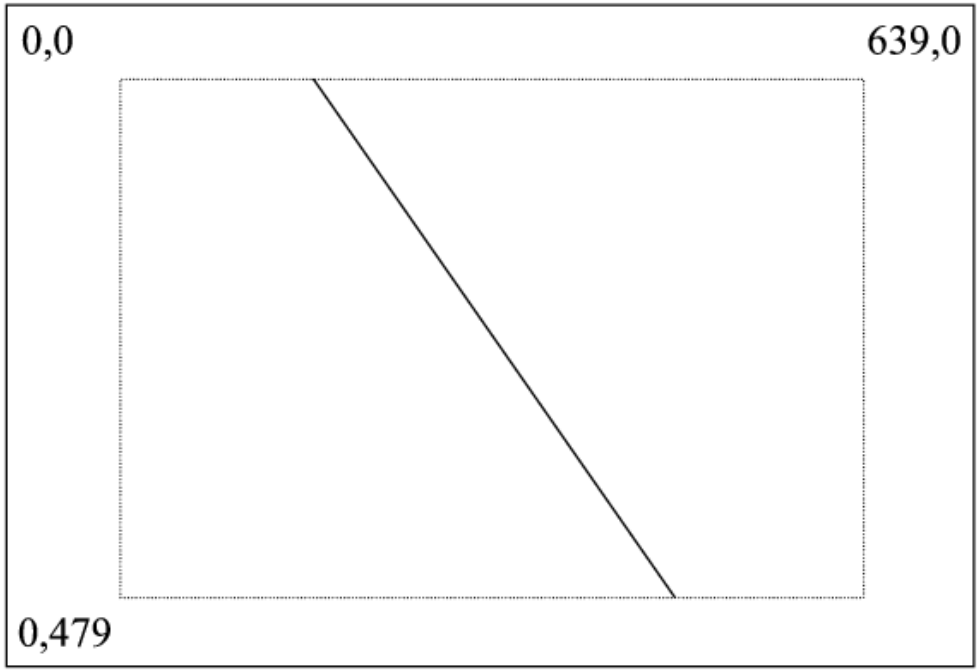
- Le type *xycoord* est défini dans la librairie *dflib*.



Systèmes de coordonnées : Clipping region

- On peut définir une région de découpage (*clipping region*), à l'extérieur de laquelle il est interdit de dessiner par l'intermédiaire de la routine setcliprgn.

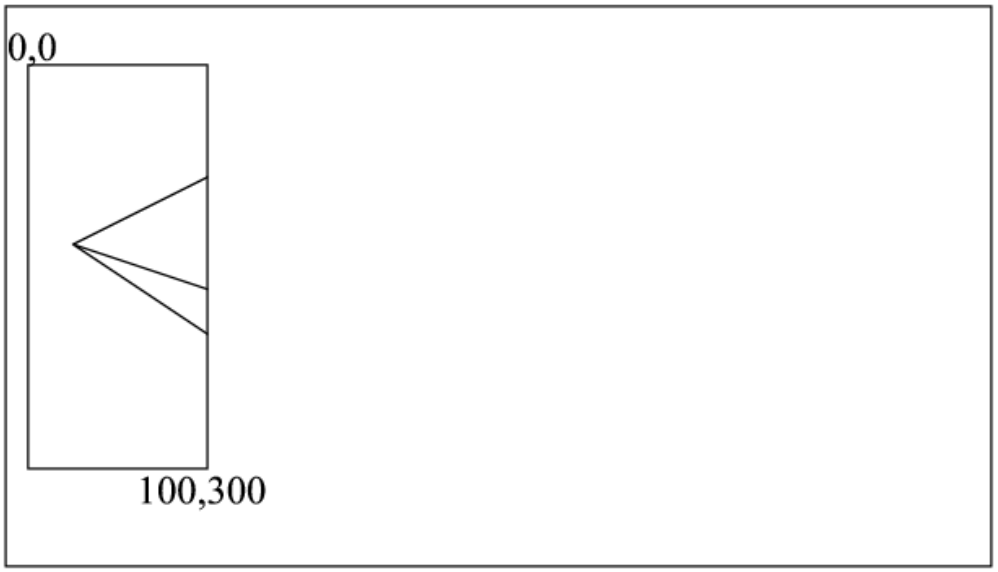
```
call setcliprgn(30_2,30_2,609_2,449_2)
```



Systèmes de coordonnées : Viewport

- La *Viewport* est la région de l'écran où l'on dessine en réalité. La routine setviewport change cette région. Elle a le même effet que stevieworg et setcliprgn combinées.

```
call setviewport(0_2,0_2,100_2,300_2)
```



La fenêtre de représentation est restreinte au rectangle sous-tendu par les points (0,0) et (100,300) par rapport au système physique.

Rq : L'origine est placée au coin supérieur gauche de la fenêtre

Systèmes de coordonnées : Coordonnées réelles (1/2)

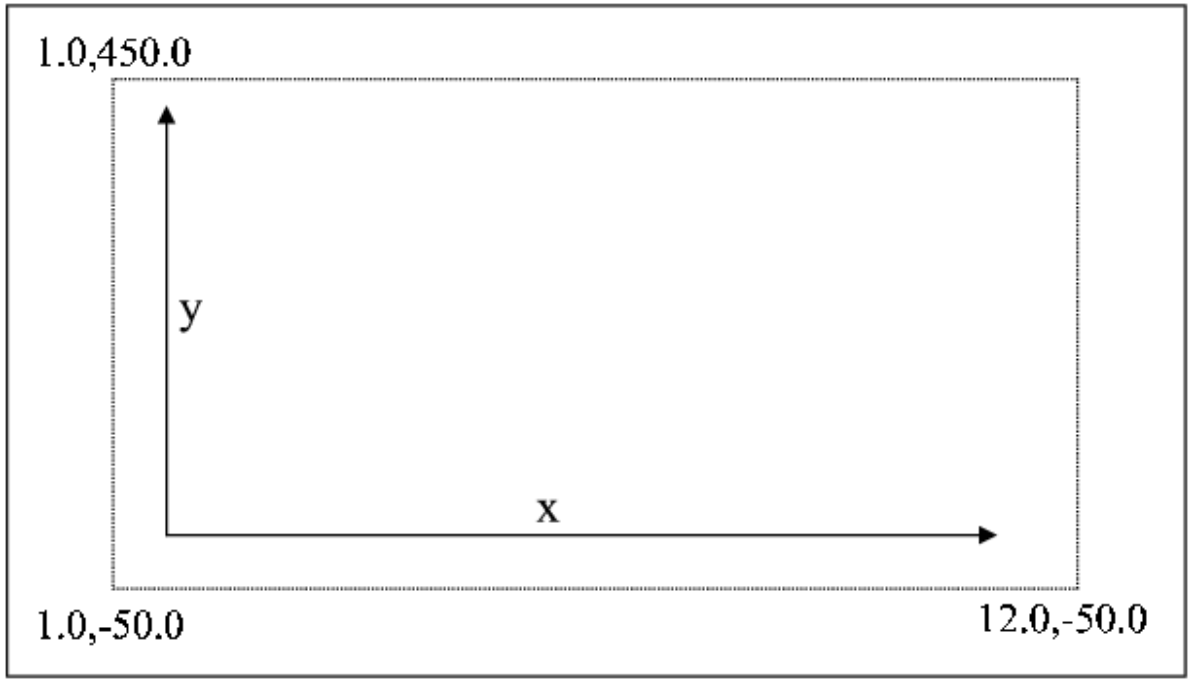
- La commande setwindow permet de changer l'échelle de la viewport en associant des « valeurs réelles » aux limites de la fenêtre de représentation.

```
dummy2 = setwindow(finvert,xmin,ymin,xmax,ymax)
```

  - finvert
    - Variable de type LOGICAL
    - Direction de l'axe y. Si .true., l'axe y augmente du bas de la fenêtre vers le haut (comme les coordonnées cartésiennes).
  - xmin,ymin
    - variable de type REAL(kind=8)
    - Coordonnées du coin inférieur gauche
  - xmax,ymax
    - variable de type REAL(kind=8)
    - Coordonnées du coin supérieur droit
  - dummy2
    - variable « muette » de type INTEGER(kind=2)
    - Différent de zéro pour sortie normale

Systèmes de coordonnées : Coordonnées réelles (2/2)

```
dummy2 = setwindow(.true.,1.0_8,-50._8,12._8,450._8)
```



Effacer l'écran

- Effacer tout l'écran :

```
call clearscreen($Gclearsreen)
```

- Effacer la *viewport* :

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

Initialisation

```
program Exemple_graphiqueini

   use dflib
   integer(kind=2) :: dummy2
   integer(kind=2),parameter:: resx=640, resy=480
   real(kind=8) :: xmin,xmax,ymin,ymax

   xmin = 0.0_8
   xmax = 10.0_8
   ymin = -1.0_8
   ymax = 1.0_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 Exemple_graphiqueini
```

Changer la couleur

- Couleur de base :

```
dummy2 = setcolor(color)
```

  - dummy2 et color sont de type INTEGER(kind=2)
  - color correspond à des valeurs prédéfinies prises entre 0 et 16.

- Couleur selon le code RGB :

```
dummy2 = setcolorrgb(color)
```

  - dummy2 est de type INTEGER(kind=2)
  - color, de type INTEGER(kind=4) 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).

```
integer(kind=4) :: blue, green, red, color
color = red + 256*green + 256*256*blue
```

red, green, blue ∈ [0,255]

Changer la couleur du fond d'écran

- Couleur de base :

```
dummy2 = setbkcolor(color)
```

  - dummy2 et color sont de type INTEGER(kind=2)
  - color correspond à des valeurs prédéfinies prises entre 0 et 16.

- Couleur selon le code RGB :

```
dummy2 = setbkcolorrgb(color)
```

  - dummy2 est de type INTEGER(kind=2)
  - color, de type INTEGER(kind=4) 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).

Relier deux points en coordonnées physiques

- Pour relier les points (x1,y1) et (x2,y2) 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)
```

- *moveto* sert à définir le point de départ.
- *lineto* relie alors (x1,y1) 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 physiques, les instructions suivantes sont nécessaires :

```
integer(kind=2) :: dummy2
real(kind=8) :: wx1, wy1, wx2, wy2
type(wxycoord) :: wxy
…
call moveto_w(x1,y1,xy)
dummy2 = lineto_w(x2,y2)
```

- *moveto_w* sert à définir le point de départ.
- *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)
```

- Remarques :
  - Control
    - $Gfillinterior pour remplir l'intérieur du rectangle
    - $Gborder pour ne dessiner que le contour
  - Les quatre derniers arguments de chaque commande définissent les deux coins opposés du rectangle.

Dessiner une ellipse

- En coordonnées physiques :

```
dummy2 = ellipse(control,x1,y1,x2,y2)
```

- En coordonnées réelles :

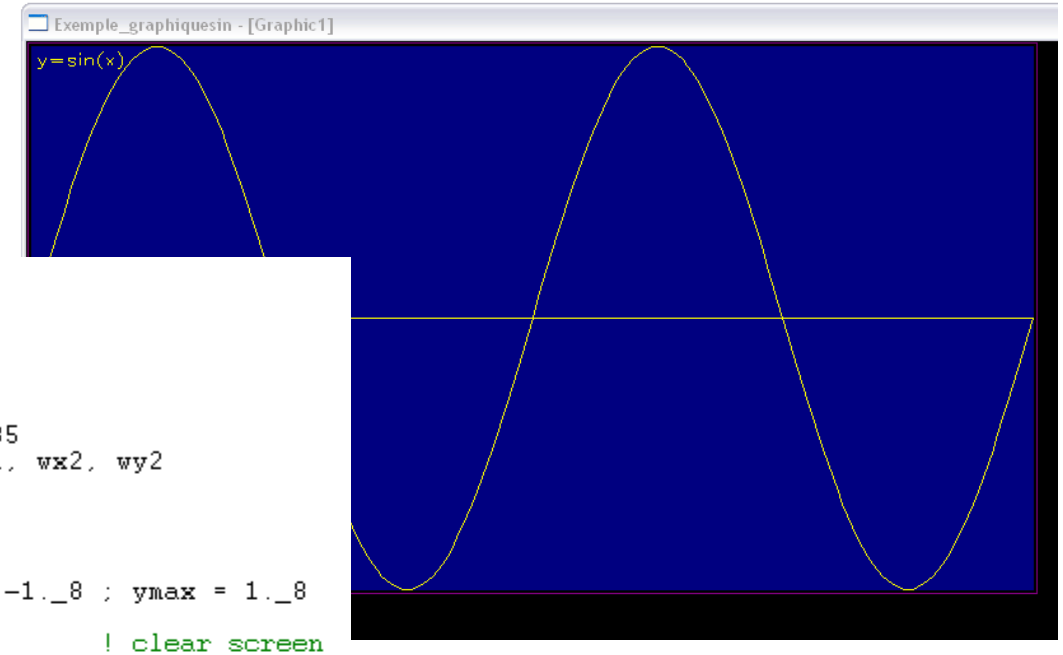
```
dummy2 = ellipse_w(control,wx1,wy1,wx2,wy2)
```

- Remarques :
  - Control
    - $Gfillinterior pour remplir l'intérieur de la forme
    - $Gborder pour ne dessiner que le contour
  - Les quatre derniers arguments de chaque commande définissent les deux extrémités de l'ellipse.

Ecriture du texte en mode graphique

```
integer(kind=2) :: dummy2, x1, y1
real(kind=8) :: wx1, wy1
type(xycoord) :: xy
type(wxycoord) :: wxy
…
dummy2 = initializefonts()              !initialiser les fonts
dummy2 = setfonts('t"modern"h14w9')     !choisir la font
call setcolor(15)                       !choisir la couleur
…
call moveto(x1,y1,xy)                  !pos en coord. physiques
call moveto_w(wx1,wy1,wxy)             !pos en coord. réelles
call outgtext("I Like Fortran 90")     !Sortie du texte
```

Exemple




```
program Exemple_graphiquesin

 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 Exemple_graphiquesin
```

Chapitre 11. Les procédures : Notions avancées

Tableaux transmis en argument : Introduction

- Un tableau peut apparaître en argument d'une procédure. Il faut pouvoir connaître le profil au sein de la procédure. Deux situations sont envisageables:

  - Le profil du tableau est connu lorsque l'on écrit la procédure.

  - Le profil du tableau n'est pas connu lorsque l'on écrit la procédure :
    - déclarer le tableau de profil implicite (profil transmis automatiquement)
    - transmettre en argument le tableau et ses étendues

Tableaux transmis en argument : Profil connu (1/2)

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

```
function produit_scalaire(vec1,vec2)
   real,dimension(1:3),intent(in) :: vec1,vec2
   real :: produit_scalaire

   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 le même profil que les dummy arguments.

Tableaux transmis en argument : Profil connu (2/2)

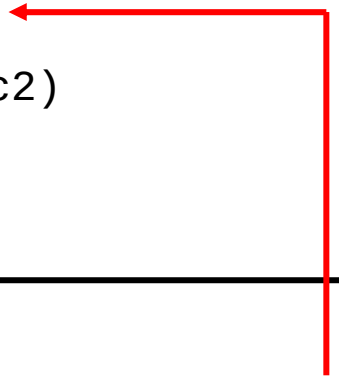
- Remarques :
  - Il est permis d'utiliser des vector subscripts pour définir les sections de tableaux à condition que la procédure ne les modifie pas.
  - Que se passe-t-il si le profil est différent de celui attendu?
    - Si l'interface de la procédure est disponible lors de la compilation (procédure interne ou bloc interface), on obtient un diagnostic de compilation
    - Dans le cas contraire, aucun diagnostic de compilation ne peut être espéré. Le risque est que la procédure travaille avec des éléments du tableau non situés à l'emplacement voulu.
  - Tableaux à allocation dynamique comme actual arguments. Ils doivent cependant :
    - être alloués avant l'appel à la procédure
    - être traités de la même manière qu'un tableau statique.

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

Tableaux transmis en argument : Profil explicite

- La deuxième 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)
   integer,intent(in) :: n
   real,dimension(1:n),intent(in) :: vec1,vec2
   real :: produit_scalaire
   real,dimension(1:2*n) :: temp
   produit_scalaire = sum(vec1*vec2)

end function produit_scalaire
```



- Le n peut être utilisé pour définir des variables locales.

Tableaux transmis en argument : Profil implicite (1/2)

- La manière la plus souple d'utiliser des tableaux comme arguments d'une procédure consiste à laisser leurs dimensions libres.

```
function produit_scalaire(vec1,vec2)
   real,dimension(:),intent(in) :: vec1,vec2
   real :: produit_scalaire

   produit_scalaire = sum(vec1*vec2)

end function produit_scalaire
```

Pour utiliser une procédure recevant en argument un tableau de profil implicite, l'interface doit être connue!

Tableaux transmis en argument : Profil implicite (2/2)

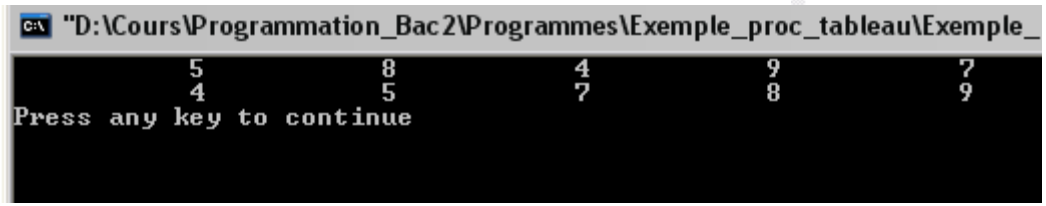
- Récupération de l'étendue au sein de la procédure
  - Possible avec l'instruction size

```
function produit_scalaire(vec1,vec2)
   real,dimension(:),intent(in) :: vec1,vec2
   real :: produit_scalaire
   real,dimension(1:2*size(vec1)) :: temp
   …
   produit_scalaire = sum(vec1*vec2)
   …
end function produit_scalaire
```

Tableaux transmis en argument : Exemple

```
program Exemple_proc_tableau1

   implicit none

   integer,parameter :: m=5
   integer,dimension(m)::tab

   interface
      subroutine tri(t,n)
      integer :: n
      integer,dimension(1:n),intent(inout) :: t
      end subroutine
   end interface

   tab = (/5,8,4,7,9/)
   write(*,*) tab
   call tri(tab,m)
   write(*,*) tab

end program Exemple_proc_tableau1

subroutine tri(t,n)

   implicit none

   integer :: n
   integer,dimension(1:n),intent(inout) :: t
   integer :: i,j
   integer ::buf

   do i = 1,n-1
      do j = i+1,n
         if (t(j)<t(i)) then
            buf = t(j)
            t(j) = t(i)
            t(i) = buf
         end if
      end do
   end do

end subroutine
```

```
program Exemple_proc_tableau2

   implicit none

   integer,parameter :: n=5
   integer,dimension(1:n) :: tab

   interface
      subroutine tri(t)
      integer,dimension(:),intent(inout) :: t
      end subroutine
   end interface

   tab = (/5,8,4,9,7/)
   write(*,*) tab
   call tri(tab)
   write(*,*) tab

end program Exemple_proc_tableau2

subroutine tri(t)

   implicit none

   integer,dimension(:),intent(inout) :: t
   integer :: i,j
   integer :: buf

   do i = 1,size(t)-1
      do j = i+1,size(t)
      if (t(j)<t(i)) then
         buf = t(i)
         t(i) = t(j)
         t(j) = buf
      end if
      end do
   end do

end subroutine
```



Les chaînes de caractères transmises en argument (1/2)

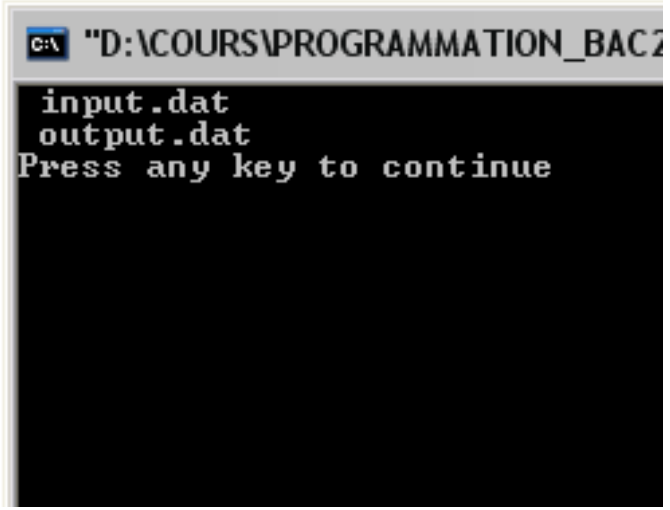
- Fortran 90 accepte qu'un argument de type chaîne de caractères possède une longueur indéterminée qu'on indique par le caractère *. La longueur effective de la chaîne sera déterminée lors de l'appel.

```
function mise_en_forme(chaine)
   character(len=*),intent(in) :: chaine
   character(len=len(chaine)+4) :: mise_en_forme


   mise_en_forme = trim(adjustl(chaine))//'.dat'


end function mise_en_forme
```

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

Les chaînes de caractères transmises en argument (2/2)

```
program Exemple_proc_chaine

   interface
      function mise_en_forme(chaine)
         character(len=*),intent(in) :: chaine
         character(len=len(chaine)+4) :: mise_en_forme
      end function mise_en_forme
   end interface

   print *, mise_en_forme('input')
   print *,mise_en_forme('output')

end program Exemple_proc_chaine

function mise_en_forme(chaine)

   character(len=*),intent(in) :: chaine
   character(len=len(chaine)+4) :: mise_en_forme

   mise_en_forme = trim(adjustl(chaine))//'.dat'

end function mise_en_forme
```



Fonctions fournissant un tableau en résultat (1/2)

- Le résultat d'une fonction peut être un tableau
- Les dimensions du tableau sont soit :
  - des constantes
  - des valeurs transmises explicitement via les arguments de la fonction
  - 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
   do i = 1,size(vecteur)
      transform(i) = i*vecteur(i)
   end do
end function transform
```

Fonctions fournissant un tableau en résultat (2/2)

```
program Exemple_proc_res_tab

   implicit none

   integer,parameter :: n = 5
   real,dimension(1:n) :: 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 Exemple_proc_res_tab

function transform(vecteur)

   real,dimension(:),intent(in) :: vecteur
   real,dimension(1:size(vecteur)) :: transform
   integer :: i

   do i = 1,size(vecteur)
      transform(i) = 3.*i*vecteur(i)
   end do

end function transform
```

```
"D:\COURS\PROGRAMMATION_BAC2\PROGRAMMES\Exemple_proc_res_tab\Debug\Exemple_..
   3.000000       6.000000       9.000000       12.00000       15.00000
Press any key to continue
```

Procédures transmises en argument (1/3)

- Notion d'argument procédure

```
subroutine sousroutine(x,y,proc)
  …
  x = 3.0 + proc(y)
  …
end subroutine sousroutine
```

```
call sousroutine(a,b,fc)
```

- Il faut faire savoir au compilateur que *fc* est effectivement une procédure. Il existe deux démarches :
  - introduire la déclaration : external fc (Fortran 77)
  - déclarer l'interface de *fc* (Fortran 90). Par cette voie, on déclare que *fc* est une fonction mais on précise aussi la nature de ses arguments (plus grande fiabilité du code).

Procédures transmises en argument (2/3)

```
function integ(xi,xf,n,f)
implicit none
real,intent(in) :: xi,xf
integer,intent(in) :: n
interface
  function f(x)
  real,intent(in) :: x
  real :: f
  end function f
end interface
real :: integ
real :: dx
integer :: i
dx = (xf-xi)/real(n)
integ = dx*(0.5*(f(xi)+f(xf)) + sum( (/(f(xi+i*dx), i=1,n-1)/) ))
end function integ
```

L'interface de la fonction *integ* doit reprendre cet ensemble de déclarations

```
function f(x)
  real,intent(in) :: x
  real :: f
  f = x*sin(x)
end function f
```

Procédures transmises en argument (3/3)

```
program Exemple_proc_arg

   interface
      function integ(xi,xf,n,f)
      real,intent(in) :: xi,xf
      integer,intent(in) :: n
      interface
         function f(x)
         real,intent(in) :: x
         real :: f
         end function f
      end interface
      real :: integ
      end function integ
      function f(x)
      real,intent(in) :: x
      real :: f
      end function f
      function g(x)
      real,intent(in) :: x
      real :: g
      end function g
   end interface

   intrinsic sin

   print *, integ(0.,3.141592,1000,f)
   print *, integ(0.,3.141592,1000,g)
   print *, integ(0.,3.141592,1000,sin)

end program Exemple_proc_arg
```

```
"D:\Cours\Programmation_Bac2\Programmes\Ex
   3.141590
   1.999999
   1.999999
Press any key to continue
```

```
function integ(xi,xf,n,f)
   implicit none

   real,intent(in) :: xi,xf
   integer,intent(in) :: n
   interface
      function f(x)
      real,intent(in) :: x
      real :: f
      end function f
   end interface
   real :: integ
   real :: dx
   integer :: i

   dx = (xf-xi)/real(n)
   integ = dx*(0.5*(f(xi)+f(xf)) + sum( (/(f(xi+i*dx), i=1,n-1)/) ))

end function integ

function f(x)

   implicit none

   real,intent(in) :: x
   real :: f

   f = x*sin(x)

end function f

function g(x)

   implicit none

   real,intent(in) :: x
   real :: g

   g = sin(x)

end function g
```

Intrinsic indique au compilateur que sin est la fonction intrinsèque connue du Fortran

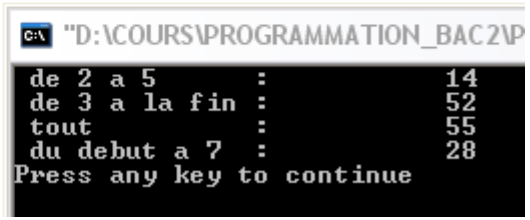
Arguments optionnels (1/2)

- Par un mécanisme approprié, il est possible à la procédure
  - de savoir si un tel argument lui a ou non été fourni lors de l'appel, et,
  - de prendre les dispositions nécessaires
- Les arguments optionnels sont déclarés avec l'attribut optional.
- L'instruction present à valeur logique permet de savoir si l'argument optionnel est présent dans l'appel à la procédure.

```
function y(x,n)
  real,intent(in) :: x
  integer,intent(in),optional :: n
  real :: y
  integer :: n$
  n$ = 1
  if (present(n)) n$ = n
  y = x**n$
end function y
```

Arguments optionnels (2/2)

```
program Exemple_arg_opt

   implicit none

   integer :: i
   integer,parameter :: n = 10
   integer,dimension(1:n) :: tab = (/ (i, i=1,n) /)
   interface
      function somme(t,deb,fin)
      integer,dimension(:),intent(in) :: t
      integer,intent(in),optional :: deb,fin
      integer :: somme
      end function somme
   end interface

   print *,'de 2 a 5       : ', somme(tab,2,5)
   print *,'de 3 a la fin  : ', somme(tab,3)
   print *,'tout           : ', somme(tab)
   print *,'du debut a 7   : ', somme(tab,fin=7)

end program Exemple_arg_opt

function somme(t,deb,fin)

   implicit none

   integer,dimension(:),intent(in) :: t
   integer,intent(in),optional :: deb,fin
   integer :: somme
   integer :: deb$,fin$

   if (present(deb)) then
      deb$ = deb
   else
      deb$ = 1
   end if

   if (present(fin)) then
      fin$ = fin
   else
      fin$ = size(t)
   end if

   somme = sum(t(deb$:fin$))

end function somme
```



```
if (.not.present(deb)) deb = 1
```

Incorrect car *deb* est déclaré avec l'attribut *intent(in)*

Procédures récursives

- On peut avoir besoin de réaliser des procédures récursives :
  - récursivité directe : une procédure comporte au moins un appel à elle-même,
  - récursivité croisée : une procédure appelle une autre procédure qui, à son tour, appelle la première (cycle pouvant comporter plus de 2 proc.).

```
recursive function fac(n) result(res)
  integer,intent(in) :: n
  integer :: res
  if (n.le.1) then
     res = 1
  then
     res = fac(n-1)*n
  end if
end function fac
```

Obligatoire dans le cas de fonctions.

Les fonctions récursives sont peu efficaces

Chapitre 12. Les modules : Notions avancées

Procédures génériques : Principe

- Si l'on considère une fonction prédéfinie telle que *abs*, on constate qu'elle peut recevoir indifféremment un argument de type *integer* ou un argument de type *real* (Chap3, p40).

  ⇒ La fonction est dite *générique*.

  A un nom unique correspondent plusieurs fonctions chacune portant un nom et une interface spécifiques.

- Cette possibilité de regrouper sous un seul nom toute une famille de procédures peut s'appliquer à des procédures définies par le développeur.

- La démarche sera :
  - définir les différentes procédures de la « famille »
  - écrire un bloc d'interfaces particulier qui spécifie le nom générique et les interfaces des différentes procédures de la famille. Cette étape sera mise en œuvre par l'intermédiaire de module.

Procédures génériques : Interfaces génériques (1/2)

```
interface sin_card
  function sin_card_sp(x)
  real(kind=4),intent(in) :: x
  real(kind=4) :: sin_card_sp
  end function sin_card_sp
  function sin_card_dp(x)
  real(kind=8),intent(in) :: x
  real(kind=8) :: sin_card_dp
  end function sin_card_dp
end interface sin_card
```

nom générique

version simple précision

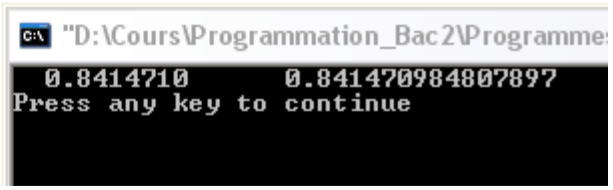
version double précision

- Cette interface peut être incluse dans la procédure qui utilise *sin_card* ou dans un module list_interface.

```
use list_interface, only: sin_card
```

Procédures génériques : Interfaces génériques (2/2)

```
module list_interface
   interface sin_card
      function sin_card_sp(x)
      real(kind=4),intent(in) :: x
      real(kind=4) :: sin_card_sp
      end function sin_card_sp
      function sin_card_dp(x)
      real(kind=8),intent(in) :: x
      real(kind=8) :: sin_card_dp
      end function sin_card_dp
   end interface sin_card
end module list_interface

program Exemple_interface_gen

   use list_interface, only : sin_card

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

end program Exemple_interface_gen

function sin_card_sp(x)

   real(kind=4),intent(in) :: x
   real(kind=4) :: sin_card_sp

   if (x.lt.1.0e-4) then
      sin_card_sp = 1.0-x**2/6.
   else
      sin_card_sp = sin(x)/x
   end if

end function sin_card_sp

function sin_card_dp(x)

   real(kind=8),intent(in) :: x
   real(kind=8) :: sin_card_dp

   if (x.lt.1.0d-4) then
      sin_card_dp = 1.0-x**2/6.d0+x**4/120.d0
   else
      sin_card_dp = sin(x)/x
   end if

end function sin_card_dp
```



Procédures génériques : Procédures de module (1/3)

```
module module_sin_card
interface sin_card
  module procedure sin_card_sp, sin_card_dp
end interface sin_card
contains
  function sin_card_sp(x)
  real(kind=4),intent(in) :: x
  real(kind=4) :: sin_card_sp
   …
  end function sin_card_sp
  function sin_card_dp(x)
  real(kind=8),intent(in) :: x
  real(kind=8) :: sin_card_dp
    …
  end function sin_card_dp
end module module_sincard
```

nom générique

version simple précision

version double précision

Procédures génériques : Procédures de module (2/3)

- On fait ensuite référence à ce module avec l'instruction

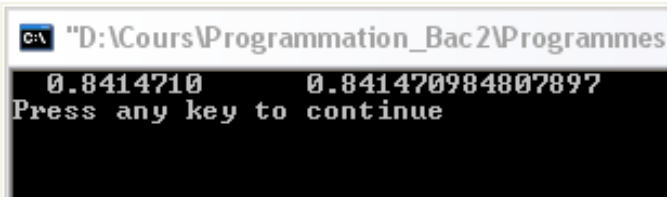
```
use module_sin_card
```

- 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.

Procédures génériques : Procédures de module (3/3)

```
module module_sin_card
   interface sin_card
      module procedure sin_card_sp,sin_card_dp
   end interface sin_card
   contains
      function sin_card_sp(x)

         real(kind=4),intent(in) :: x
         real(kind=4) :: sin_card_sp

         if (x.lt.1.0e-4) then
            sin_card_sp = 1.0-x**2/6.
         else
            sin_card_sp = sin(x)/x
         end if

      end function sin_card_sp

      function sin_card_dp(x)

         real(kind=8),intent(in) :: x
         real(kind=8) :: sin_card_dp

         if (x.lt.1.0d-4) then
            sin_card_dp = 1.0-x**2/6.d0+x**4/120.d0
         else
            sin_card_dp = sin(x)/x
         end if

      end function sin_card_dp

end module module_sin_card

program Exemple_moduleproc

   use module_sin_card

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

end program Exemple_moduleproc
```



Définition d'opérateurs : Principe

- En Fortran 90, il devient possible de :
  - donner une signification à tout symbole opératoire existant (+,-,*,/,<,>,.and.,…) lorsqu'il porte sur des types différents de ceux pour lesquels il est déjà défini.
  - créer de nouveaux opérateurs (de la forme .op. où *op* désigne une suite quelconque de caractères).

- Dans les deux cas, on utilise le même mécanisme, à savoir qu'on définit une fonction générique de nom *operator* (op), *op* désignant l'opérateur concerné (+, *, .and., .plus.,…) avec :
  - deux arguments (*intent(in)*), s'il s'agit d'un opérateur à deux opérandes
  - un argument (*intent(in)*) s'il s'agit d'un opérateur à un opérande.

Définition d'opérateurs : Implémentation

```
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(3)
  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 : Utilisation

- Pour utiliser l'opérateur .cross., il suffit alors d'écrire

```
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 : Exemple

```
module kinds
   integer,parameter :: r=8
end module kinds


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(3)
      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

program Exemple_moduleop

   use kinds
   use module_cross
   implicit none

   real(kind=r),dimension(1:3) :: a,b


   a = (/ 1._r, 0._r, 0._r /)
   b = (/ 0._r, 1._r, 0._r /)

   print *, a.cross.b


end program Exemple_moduleop
```

```
"D:\COURS\PROGRAMMATION_BAC2\PROGRAMMES\Exemple_moduleop\Debug\Exemple_
  0.00000000000000000E+000  0.00000000000000000E+000   1.00000000000000
Press any key to continue
```

Définition d'opérateurs : Extension d'opérateurs (1/2)

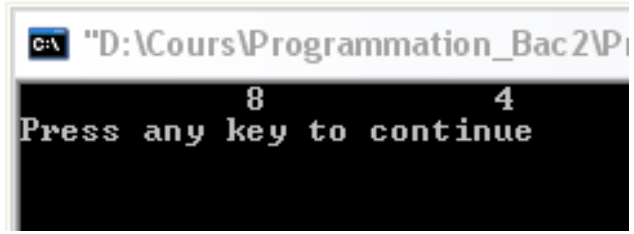
- On insistera à nouveau sur le fait qu'on peut étendre la portée d'opérateurs intrinsèques du Fortran.

- Par contre, on ne peut 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.

Définition d'opérateurs : Extension d'opérateurs (2/2)

```
module type_point

   implicit none
   type point
      integer :: x,y
   end type point
   interface operator (+)
      module procedure point_plus_point
   end interface
   contains
      function point_plus_point(p1,p2)

      type(point),intent(in) :: p1,p2
      type(point) :: point_plus_point

      point_plus_point = point(p1%x+p2%x, p1%y+p2%y)

      end function point_plus_point
end module type_point

program Exemple_surdef_op

   use type_point
   implicit none

   type(point) :: a = point(3,2), b = point(5,2), c

   c = a + b

   print *, c

end program Exemple_surdef_op
```



Surdéfinition de l'affectation

- Il est également possible d'étendre la portée du signe d'affectation (=) ou de créer ses propres opérations d'affectations.

- On utilise pour cela un bloc d'*interface assignment* (=).

- Ici aussi, il n'est pas permis de redéfinir des cas déjà implémentés.

Privatisation des ressources d'un module

- Par défaut, toutes les variables définies dans un module sont accessibles par les procédures qui y font appel.

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

```
module atomic_units
  use kinds
  real(kind=r),parameter,private :: ec = 1.602189E-19_r, &
                                    em = 9.10953E-31_r, &
                                    hbar = 1.054589E-34_r, &
                                    eps0 = 8.85418782E-12_r, &
  real(kind=r),parameter :: rBohr = 4._r*pi*eps0*hbar**2/(em*ec**2)
end module atomic_units
```

Chapitre 13. Les pointeurs

Notion de pointeurs

- Définition :
  - Un *pointeur* ne contient pas de données mais *« pointe » vers* une variable (scalaire ou tableau) où la donnée est stockée, une *cible*.
  - En Fortran, le pointeur est un *alias* contrairement au Pascal ou C où le pointeur indique *l'adresse mémoire de l'objet*.
  - Un pointeur n'a *pas de stockage initial* (réservation mémoire). La réservation est établie quand le programme s'exécute.

- Etats d'un pointeur :
  - Indéfini $\Rightarrow$ A la déclaration en tête de programme
  - Nul $\Rightarrow$ Alias d'aucun objet
  - Associé $\Rightarrow$ Alias d'un objet appelé cible

Déclaration des pointeurs et des cibles

- Les pointeurs et les cibles doivent être déclarés avec le même type, le même kind et le même rang.
- En ce qui concerne les attributs :
  - Les pointeurs reçoivent l'attribut : pointer
  - Les cibles reçoivent l'attribut : target

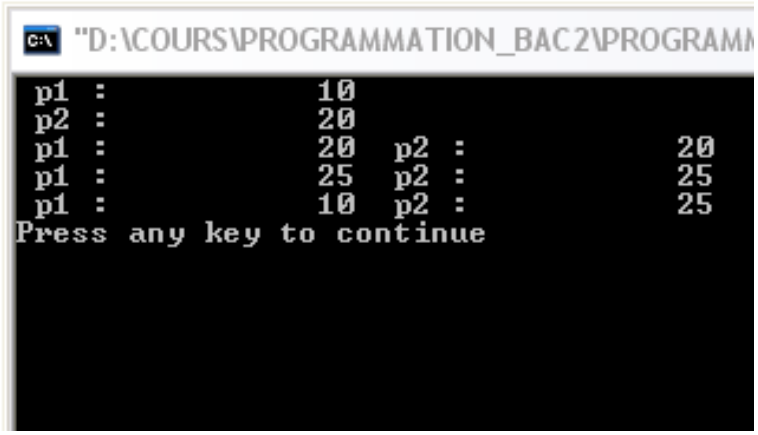
```
integer,pointer :: p1
real(kind=r),dimension(:),pointer :: p2
character(len=80),dimension(:),pointer :: p3
type boite
   integer :: i
   character(len=5) :: t
end type
type(boite),pointer :: p4
integer, target :: n
```

Symbole => (1/2)

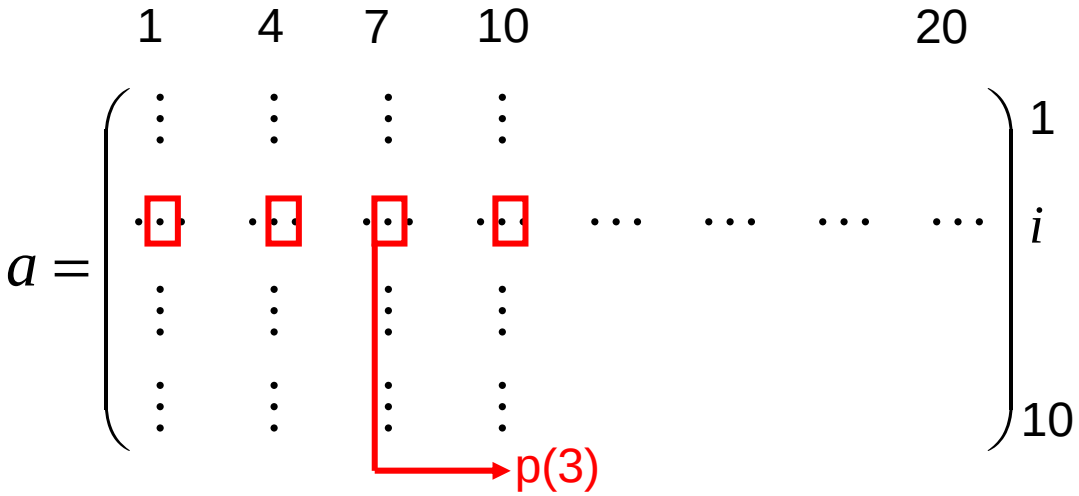
- Le symbole => sert à affecter une valeur à un pointeur.

`p1 => p2` → `p1 prend l'état de p2`

```
program Exemple_pointeur

   implicit none

   integer, pointer :: p1,p2
   integer, target :: n,m

   n = 10
   m = 20

   p1 => n
   print *, 'p1 : ',p1
   p2 => m
   print *, 'p2 : ',p2
   p1 => p2
   print *, 'p1 : ',p1,' p2 : ', p2
   m = 25
   p2 => m
   print *, 'p1 : ',p1,' p2 : ', p2
   p1 => n
   print *, 'p1 : ',p1,' p2 : ', p2

end program Exemple_pointeur
```



Symbole => (2/2)




```
program Exemple_pointeurb

    implicit none

    real,dimension(10,20),target :: a
    real,dimension(:),pointer :: p
    integer :: i

    a = reshape( (/ (real(i), i=1,200) /), (/ 10 ,20 /) )

    print *,'Entrez un numero de colonne :'
    read(*,*) i

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

end program Exemple_pointeurb
```

```
"D:\Cours\Programmation_Bac2\Programmes\Exemple_pointeur\
 Entrez un numero de colonne :
5
   65.00000
Press any key to continue
```

Symbole = appliqué aux pointeurs

- Lorsque les deux opérandes de = sont des pointeurs, l'affectation s'effectue sur les cibles et non sur les pointeurs.

```
program Exemple_pointeur1

   implicit none

   integer :: i
   real,dimension(3,3),target :: a,b
   real,dimension(:,:),pointer :: p1,p2

   a = reshape( (/ (real(i), i=1,9) /), (/ 3,3 /) )

   p1 => a
   p2 => b

   p2 = p1 + 1.
   print *,b

end program Exemple_pointeur1
```

```
"D:\Cours\Programmation_Bac2\Programmes\Exemple_pointeur\Exemple_pointeur1\Debug
   2.000000       3.000000       4.000000       5.000000       6.000000
   7.000000       8.000000       9.000000       10.00000
Press any key to continue
```

Allocation dynamique de la mémoire

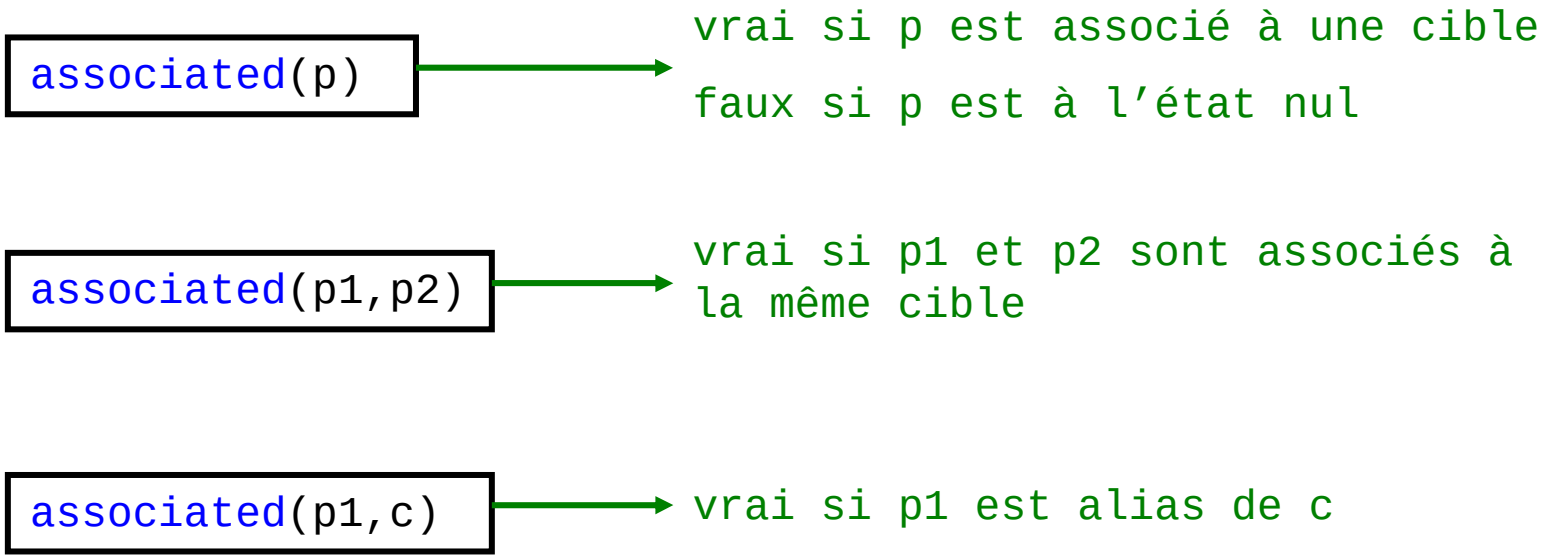
- L'instruction allocate permet d'associer un pointeur et d'allouer dynamiquement de la mémoire.
- L'instruction deallocate permet de libérer la place allouée.

```
character(len=80),pointer :: p
character(len=80) :: ch1, ch2
…
if (ch1.gt.ch2) then
  allocate(p)
  p = ch1
  ch1 = ch2
  ch2 = p
  deallocate(p)
end if
```

Interrogation et comparaison de pointeurs

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

```
associated(<pointeur1> [,<pointeur2>])
associated(<pointeur>,<cible>)
```



Instruction NULLIFY

- Au début d'un programme un pointeur n'est pas défini, son état est indéterminé. L'instruction nullify permet de forcer un pointeur à l'état nul.

```
real,pointer :: p1,p2
nullify(p1)
nullify(p2)
```

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

Chapitre 14. Notions élémentaires d’optimisation

Motivation

- L'optimisation est la pratique qui consiste généralement à réduire :
  - le temps d'exécution d'une fonction,
  - l'espace occupé par les données et le programme.

- On peut optimiser à plusieurs niveaux un programme :
  - Algorithme : choisir un algorithme de complexité inférieure au sens mathématique et des structures de données adaptées.

    $\Rightarrow$ Cours d'approche numérique Bac3
  - Langage : ordonner au mieux les instructions et utiliser les bibliothèques disponibles.

- Règle 0 :
  - L'optimisation (Langage) *ne doit intervenir qu'une fois que le programme fonctionne et répond aux spécifications fonctionnelles*.

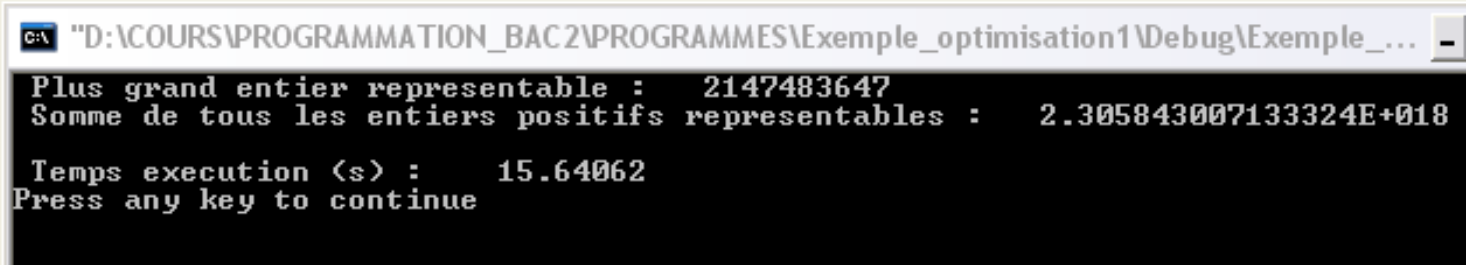
Mesurer le temps d'exécution (1/2)

- Avant de commencer l'optimisation, il faut savoir mesurer la vitesse du code.

```
program main
real(kind=4) :: time1,time2
call cpu_time(time1)
…
call cpu_time(time2)
print *, 'temps execution (s) : ', time2-time1
end program main
```

Mesurer le temps d'exécution (2/2)

```
program Exemple_optimisation1

   implicit none

   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 = 0.0d0
      do i = 1,imax
         x = x + real(i,8)
      end do
   end do

   print *, 'Plus grand entier representable : ', imax
   print *, 'Somme de tous les entiers positifs representables : ', x
   print *

   call cpu_time(time2)
   print *, 'Temps execution (s) : ', time2-time1

end program Exemple_optimisation1
```



Comment accélérer un programme ?

- Au niveau langage, on peut se tenir à trois grands principes :
  - Choisir les options de compilation
  - Minimiser le nombre d'opérations
  - Gérer de manière optimale l'accès à la mémoire

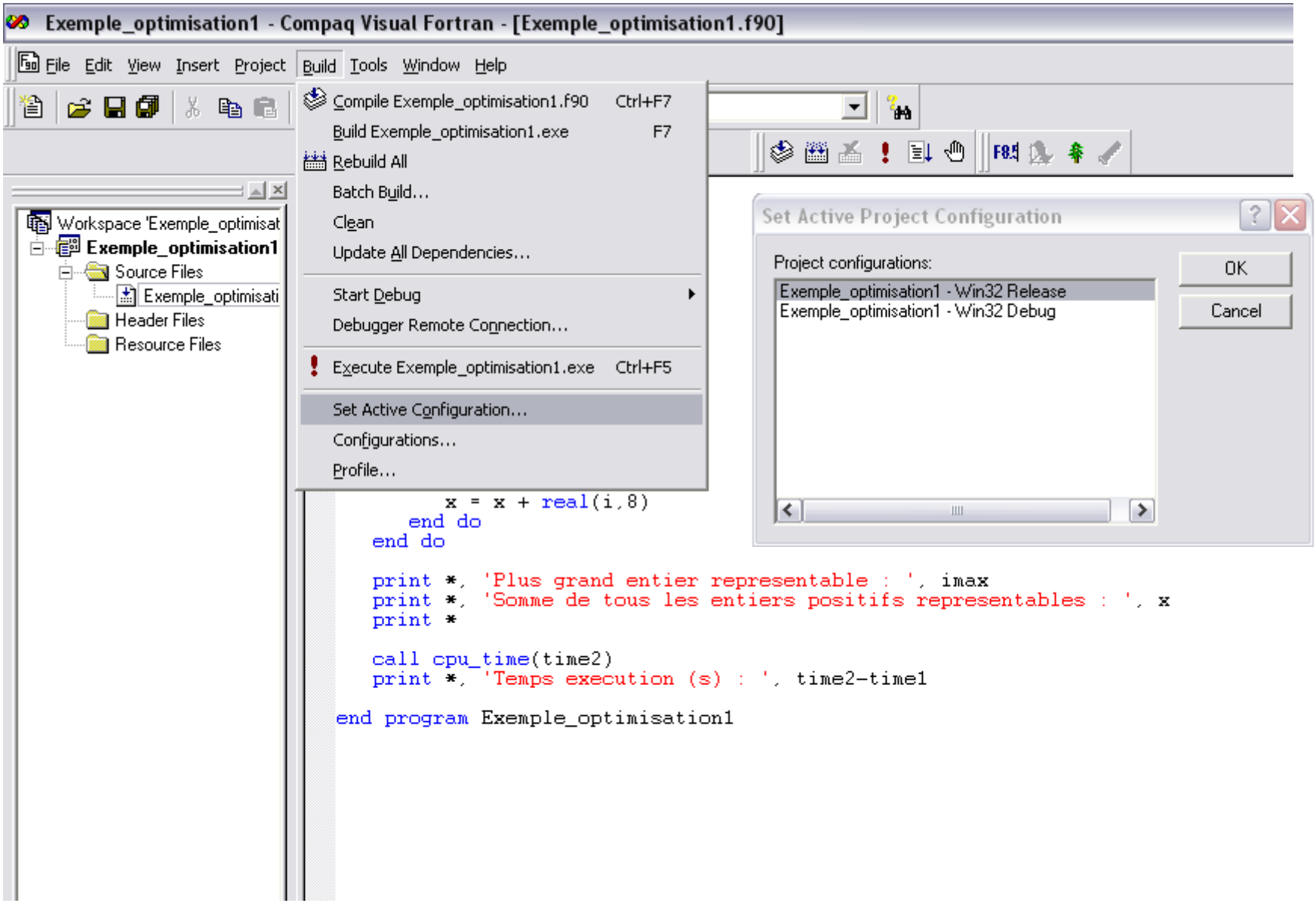
Choix des options de compilation (1/3)

- Les compilateurs sont souvent capable de faire des optimisations locales.

- Avec Developer Studio, deux possibilités :
  - Aller dans Build\Set Active Configuration $\Rightarrow$ Choisir *Release* plutôt que *Debug*.

  - Aller dans Project\Settings\Fortran\Category:Optimizations où des options plus avancées s'y trouvent.

Choix des options de compilation (2/3)



Choix des options de compilation (3/3)

```
program Exemple_optimisation1

   implicit none

   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 = 0.0d0
      do i = 1,imax
         x = x + real(i,8)
      end do
   end do

   print *, 'Plus grand entier representable : ', imax
   print *, 'Somme de tous les entiers positifs representables : ', x
   print *

   call cpu_time(time2)
   print *, 'Temps execution (s) : ', time2-time1

end program Exemple_optimisation1
```

```
"D:\COURS\PROGRAMMATION_BAC2\PROGRAMMES\Exemple_optimisation1\Release\Exemple...
 Plus grand entier representable :   2147483647
 Somme de tous les entiers positifs representables :   2.305843007133324E+018

 Temps execution (s) :    6.859375
Press any key to continue
```

Minimisation des opérations (1/2)

- Le deuxième grand principe consiste à :
  - Nettoyer les boucles

⇒ Sortir des boucles les calculs qui peuvent être réalisés en dehors de celles-ci.

  - Définir des variables intermédiaires

⇒ 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.

Minimisation des opérations (2/2)

```
program Exemple_optimisation2

   implicit none

   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
   end do
      x = 0.0d0
   do i = 1,imax
      x = x + real(i,8)
   end do

   print *, 'Plus grand entier representable : ', imax
   print *, 'Somme de tous les entiers positifs representables : ', x
   print *

   call cpu_time(time2)
   print *, 'Temps execution (s) : ', time2-time1

end program Exemple_optimisation2
```

```
"D:\Cours\Programmation_Bac2\Programmes\Exemple_optimisation1\Exemple_optimisation...
 Plus grand entier representable :   2147483647
 Somme de tous les entiers positifs representables :   2.305843007133324E+018

 Temps execution (s) :    3.640625
Press any key to continue
```

Gestion de l'accès à la mémoire (1/2)

- Le troisième grand principe consiste à accéder aux éléments d'un tableau dans leur ordre naturel.

```
matrice(1,1)
matrice(2,1)
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)…
  end do
end do
```

inefficace

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

L'objectif est de réduire le temps d'accès à la mémoire.

Gestion de l'accès à la mémoire (2/2)

Ecriture	Debug	Release

`Ax = 0.0`
`do i = 1,n`
`  do j =1,n`
`  Ax(i) = Ax(i)+A(i,j)*x(j)`
`  end do`
`end do`
	1.20 s	0.91 s

`Ax = matmul(A,x)`	1.04 s	0.89 s

`Ax = 0.0`
`do j = 1,n`
`  do i =1,n`
`  Ax(i) = Ax(i)+A(i,j)*x(j)`
`  end do`
`end do`
	0.86 s	0.31 s

`Ax = 0.0`
`do j =1,n`
`  Ax(:) = Ax(:)+A(:,j)*x(j)`
`end do`
	0.74 s	0.31 s

n = 8000

Chapitre 15. Exécution en lignes de commandes

Ouvrir une fenêtre d'invite de commandes

- Sous Windows :
  - Démarrer\(Tous les)Programmes\Accessoires\Invite de commandes
  - Démarrer\Exécuter ⇒ cmd

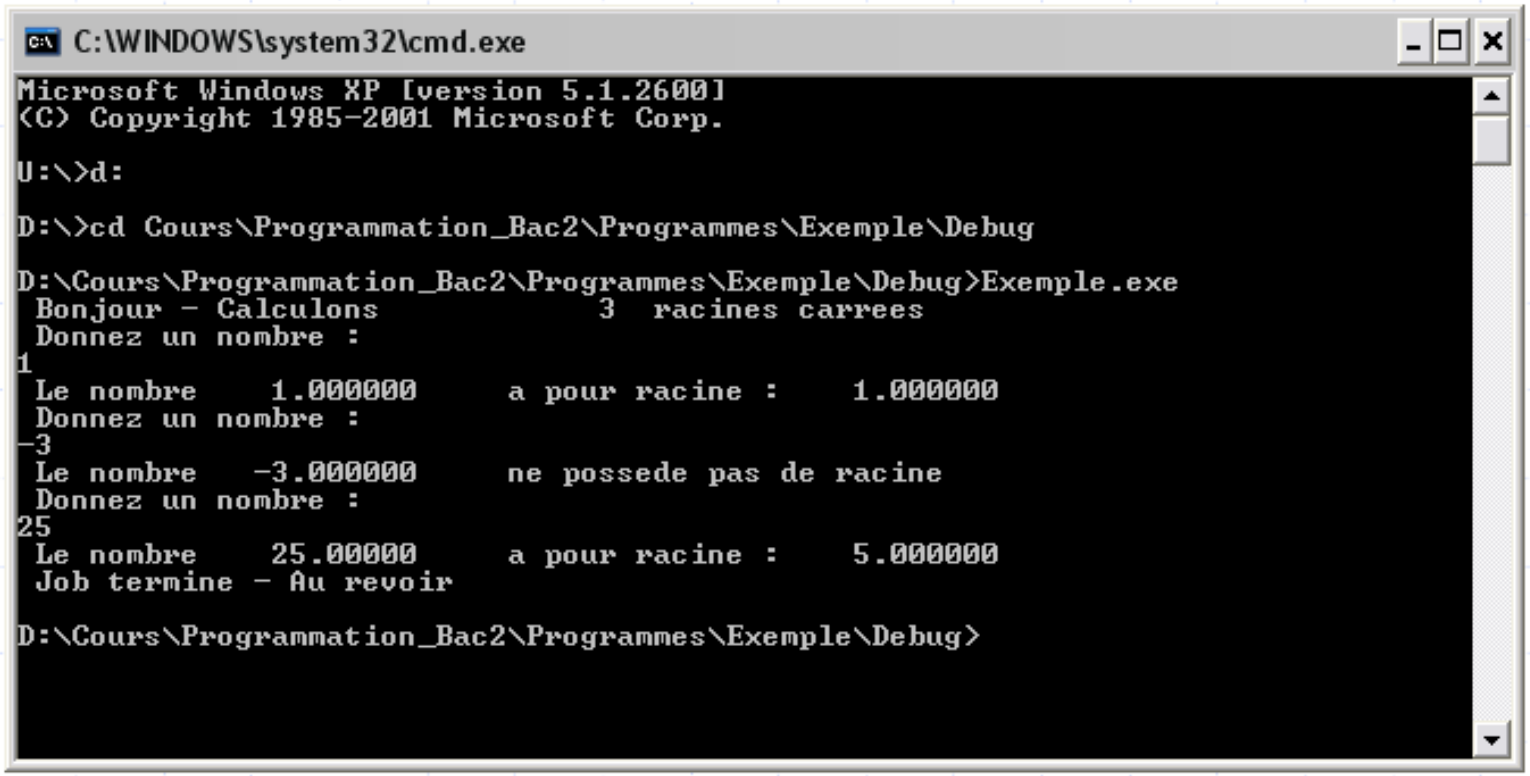
Instructions de base DOS

- c:, d:, … : changer de disque
- cd : changer de répertoire
  - cd dir1 : aller dans le répertoire *dir1*
  - cd dir1\dir2 : aller dans le répertoire *dir1*, ensuite dans le répertoire *dir2*
  - cd .. : remonter d'un niveau dans l'arborescence des répertoires
- dir : affiche le contenu du répertoire courant
- mkdir : créer un répertoire
- copy : copier des fichiers
- move : déplacer des fichiers ou des répertoires
- del : effacer des fichiers
- help : obtenir l'aide sur les instructions disponibles et leurs options

  (Exemple : help, help dir)

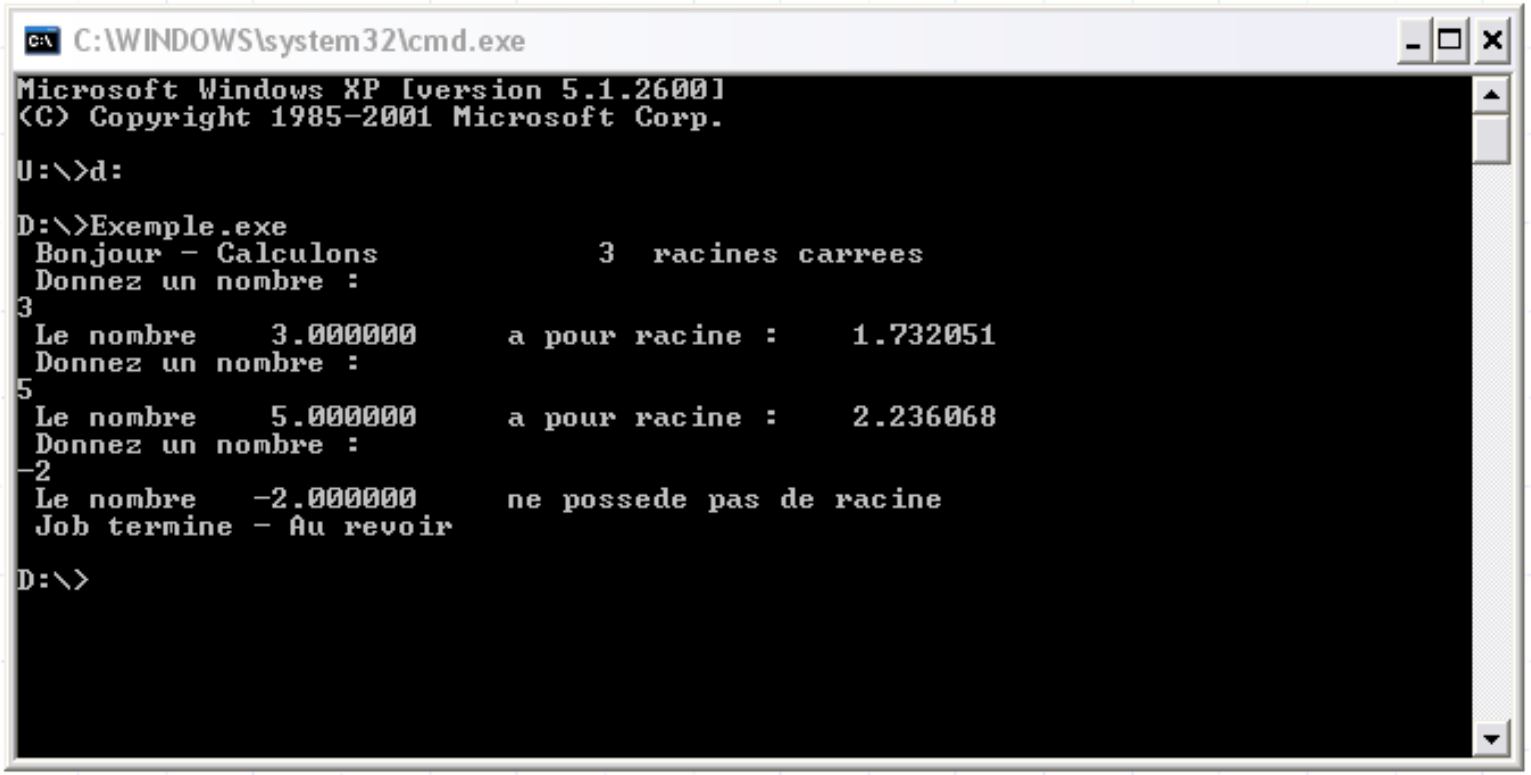
Exécution : Répertoire courant

- On peut lancer un exécutable en lignes de commandes en écrivant son nom. Cet exécutable doit se trouver dans le répertoire courant.



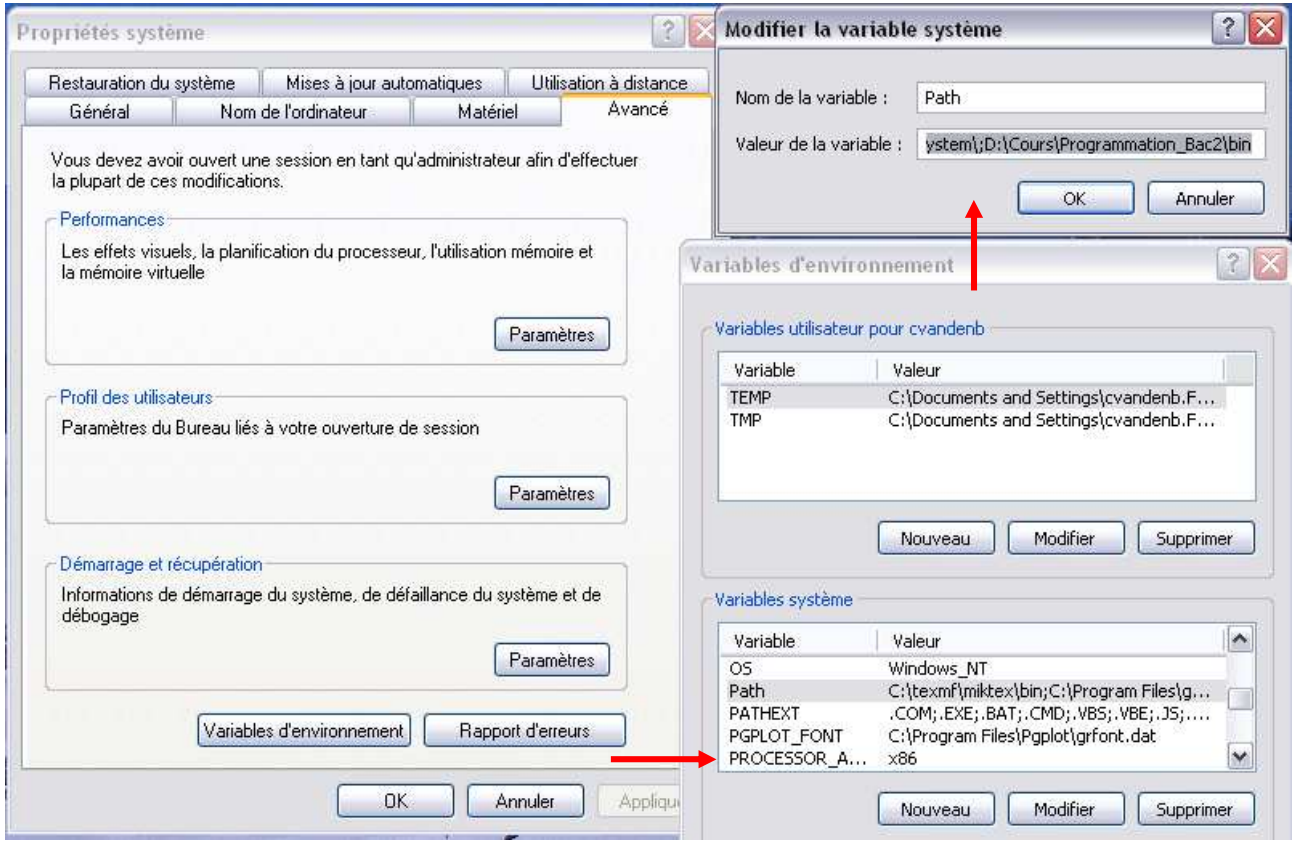
Exécution : Répertoire indiqué par PATH (1/3)

- On peut lancer un exécutable en lignes de commandes en écrivant son nom. Cet exécutable doit se trouver dans le répertoire indiqué par la variable d'environnement PATH.



Exécution : Répertoire indiqué par PATH (2/3)

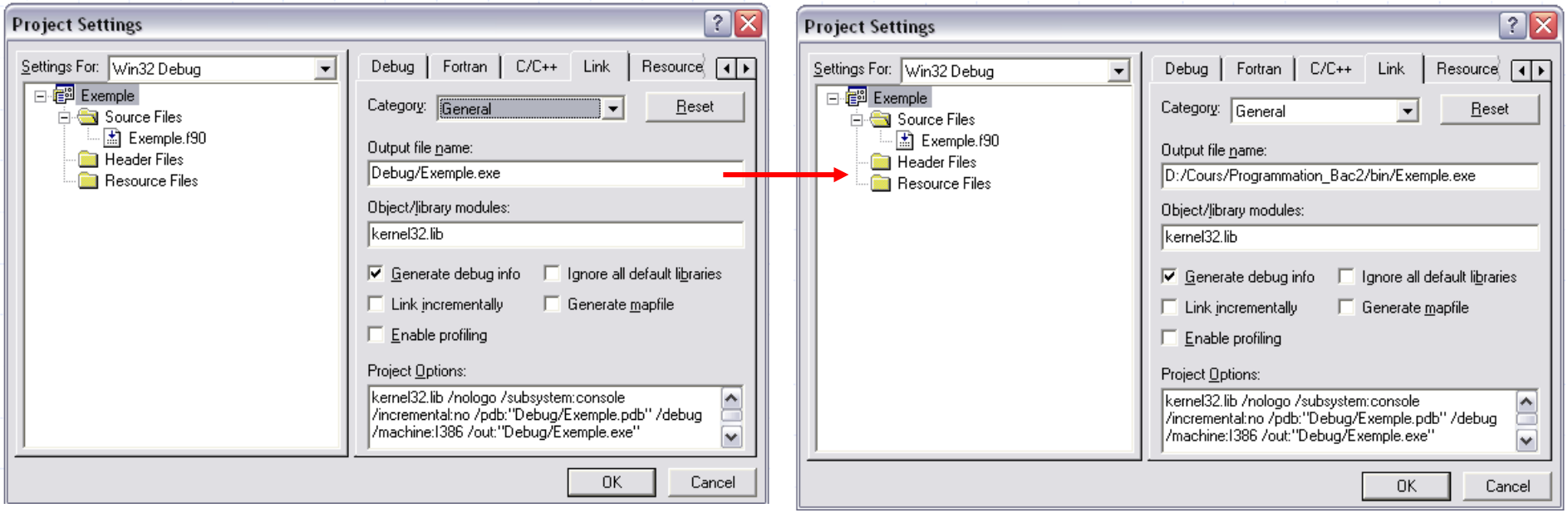
- Pour modifier cette variable d'environnement:
  - Poste de Travail\[clic droit]\Propriétés
  - Avancé
  - Variables d'environnement
  - Sélectionner PATH
  - Modifier…



Exécution : Répertoire indiqué par PATH (3/3)

- Vous pouvez indiquer dans le Developer Studio le répertoire destiné à recevoir l'exécutable.

  - Project\Settings…\Link\Output file name :



Récupération d'arguments (1/2)

- Nombre d'arguments :

```
<result> = nargs()
```

  - Result :
    - variable de type INTEGER(4) qui retourne le nombre d'arguments en incluant la commande elle-même. Par exemple, *nargs* retourne 4 pour PROG –g –c –a

- Argument de la ligne de commande :

```
call getarg(<n>,<buffer> [,<status>])
```

  - n :
    - variable de type INTEGER(2) qui donne la position de la commande à rechercher. La commande elle-même est l'argument 0.
  - buffer :
    - variable de type CHARECTER(*) qui donne l'argument recherché.
  - status :
    - variable de type INTEGER(2) qui donne le nombre de caractères de l'argument. En cas d'erreur, *status* vaut -1.

Récupération d'arguments (2/2)

```
program Exemple_arg_lignecom

   use dflib
   implicit none

   integer, parameter :: numfich=10
   character(len=12) :: nomfich
   integer :: ierr
   character(len=1) :: c
   integer :: narg

   narg = nargs()-1
   if (narg.gt.0) then
      call getarg(1,nomfich)
   else
      write(*, '("nom du fichier : ")',advance='no')
      read(*,*) nomfich
      write(*,*)
   end if

!Lecture du fichier et affichage à l'écran
   open(numfich,file=nomfich)
   do
      read(numfich,'(a1)',advance='no', eor=88, end=99) c
      write(*,'(a1)',advance='no') c
      cycle
88    write(*,*)    !forcer un changement de li
      cycle
99    exit
   end do
   write(*,'(//"-- fin du fichier")')
   close(numfich)

end program Exemple_arg_lignecom
```

```
C:\WINDOWS\system32\cmd.exe

D:\Cours\Programmation_Bac2\bin>Exemple_arg_lignecom.exe
nom du fichier : Fortran.txt

Fortran (previously FORTRAN) is a general-purpose,
procedural, imperative programming language that
is especially suited to numeric computation and
scientific computing. Originally developed by IBM in
the 1950s for scientific and engineering applications,
Fortran came to dominate this area of programming
early on and has been in continual use for over half
a century in computationally intensive areas such as
numerical weather prediction, finite element analysis,
computational fluid dynamics (CFD), computational
physics, and computational chemistry. It is one of the
most popular languages in the area of high-performance
computing and is the language used for programs that
benchmark and rank the world's fastest supercomputers.


-- fin du fichier
```

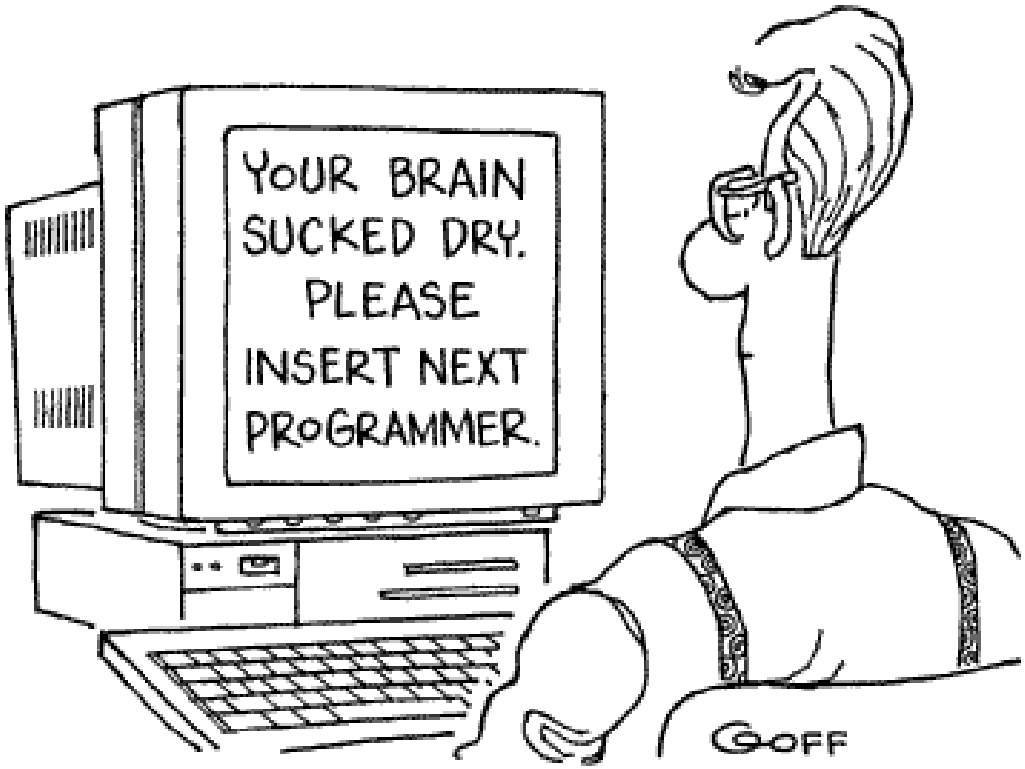
```
C:\WINDOWS\system32\cmd.exe

D:\Cours\Programmation_Bac2\bin>Exemple_arg_lignecom.exe Fortran.txt
Fortran (previously FORTRAN) is a general-purpose,
procedural, imperative programming language that
is especially suited to numeric computation and
scientific computing. Originally developed by IBM in
the 1950s for scientific and engineering applications,
Fortran came to dominate this area of programming
early on and has been in continual use for over half
a century in computationally intensive areas such as
numerical weather prediction, finite element analysis,
computational fluid dynamics (CFD), computational
physics, and computational chemistry. It is one of the
most popular languages in the area of high-performance
computing and is the language used for programs that
benchmark and rank the world's fastest supercomputers.


-- fin du fichier
```

Appel de commandes DOS en Fortran

- L'instruction call system("<commande>") permet d'exécuter des commandes DOS en Fortran.

```
call system("  dir > fichier.dat")
call system ("Graph.exe")
```

Fin



MCours.com