

UPMC

Master P&A/SDUEE

UE MNI (4P009)

Méthodes Numériques et Informatiques

Fortran 90/95 et C

`Sofian.Teber@lpthe.jussieu.fr`

`Jacques.Lefrere@upmc.fr`

Albert Hertzog

2014–2015

Table des matières

1	Introduction	16
1.1	Programmation en langage compilé	16
1.2	Historique	19
1.2.1	Langage fortran	19
1.2.2	Langage C	20
1.3	Intérêts respectifs	21
1.4	Généralités	23
1.5	Exemple de programme C avec une seule fonction utilisateur	24
1.6	Exemple de programme fortran avec une seule procédure	25
1.7	Structure générale d'un programme C élémentaire	26
1.8	Exemple de programme C avec deux fonctions	28

1.9	Structure générale d'un programme fortran élémentaire	30
1.10	Ex. de programme fortran avec fonction dans un module	33
1.11	Compilation et édition de liens	35
1.12	Compilateurs utilisés	38
2	Types et déclarations des variables	39
2.1	Types de base	39
2.2	Domaine et précision	41
2.2.1	Domaine des entiers signés	41
2.2.2	Valeurs maximales des entiers en C	44
2.2.3	Domaine et précision des flottants	47
2.2.4	Caractéristiques des types numériques en fortran	51
2.3	Déclarations des variables	53

2.4	Exemples d'attribut constante	55
2.4.1	Exemples d'attribut <code>const</code> en C	55
2.4.2	Exemples d'attribut <code>parameter</code> en fortran	55
2.5	Les constantes	57
3	Opérateurs	58
3.1	Opérateur d'affectation	58
3.2	Opérateurs algébriques	60
3.3	Opérateurs de comparaison	60
3.4	Opérateurs logiques	61
3.5	Incrémentation et décrémentation en C	63
3.6	Opérateurs d'affectation composée en C	64
3.7	Opérateur d'alternative en C	64

3.8	Opérateurs agissant sur les bits	65
3.9	Opérateur sizeof en C	66
3.10	Opérateur séquentiel «,» en C	66
3.11	Opérateurs & et * en C	66
3.12	Priorités des opérateurs en C	67
4	Entrées et sorties standard élémentaires	68
4.1	Introduction aux formats d'entrée–sortie	69
4.1.1	Introduction aux formats en C	72
5	Structures de contrôle	74
5.1	Structure conditionnelle <code>if</code>	75
5.1.1	Condition <code>if</code>	75
5.1.2	Alternative <code>if ... else</code>	75

5.1.3	Exemples d'alternative <code>if ... else</code>	77
5.1.4	Alternatives imbriquées <code>if ... else</code>	79
5.1.5	Cas de <code>else if</code> en fortran	80
5.2	Aiguillage avec <code>switch/case</code>	81
5.2.1	Exemples d'aiguillage <code>case</code>	82
5.3	Structures itératives ou boucles	87
5.3.1	Exemples de boucle <code>for</code> ou <code>do</code>	90
5.4	Branchements ou sauts	93
5.4.1	Exemples de bouclage anticipé <code>cycle/continue</code>	95
5.4.2	Exemple de sortie de boucle <code>break/exit</code>	96
6	Introduction aux pointeurs	99
6.1	Intérêt des pointeurs	99

6.2	Pointeurs et variables : exemple du C	100
6.2.1	Affectation d'un pointeur en C	103
6.2.2	Indirection (opérateur * en C)	107
6.3	Pointeurs en fortran	109
6.4	Syntaxe des pointeurs (C et fortran)	110
6.5	Exemples élémentaires (C et fortran)	111
6.6	Tailles des types de base et adresses en C	113
6.7	Initialiser les pointeurs !	115
7	Procédures : fonctions et sous-programmes	118
7.1	Généralités	118
7.1.1	Structure d'un programme C	120
7.1.2	Structure générale d'un programme fortran	122

7.1.3	Fonctions renvoyant une valeur	124
7.1.4	Sous-programmes et fonctions C sans retour	127
7.1.5	Passage des arguments : nécessité des pointeurs en C	128
7.1.6	Vocation des arguments en fortran	129
7.2	Exemples de procédures	130
7.2.1	Faux échange en C	130
7.2.2	Vrai échange avec pointeurs en C	133
7.2.3	Procédure de module en fortran 90	136
7.3	Durée de vie et portée des variables	138
7.3.1	Exemples de mauvais usage des variables locales	141
7.3.2	Exemples de variable locale permanente	143
7.3.3	Exemples d'usage de variable globale	145
7.4	Visibilité des interfaces et compilation séparée	147

7.5	Compléments sur les procédures	150
7.5.1	Exemples de procédures récursives : factorielle	151
7.5.2	Les fonctions mathématiques	160
7.5.3	Une erreur classique : la fonction <code>abs</code> en flottant en C	164
8	Tableaux	165
8.1	Définition et usage	165
8.2	Tableaux de dimension fixe	166
8.3	Tableaux en fortran	168
8.3.1	Opérations globales sur les tableaux en fortran	169
8.3.2	Sections régulières de tableaux en fortran	169
8.3.3	Fonctions opérant sur des tableaux en fortran	170
8.3.4	Parallélisation en fortran	173

8.3.5	Ordre des éléments de tableaux 2D en fortran	173
8.4	Tableaux et pointeurs et en C	177
8.4.1	Ordre des éléments de tableaux 2D en C	178
8.4.2	Sous-tableau 1D avec un pointeur	181
8.4.3	Utilisation de <code>typedef</code>	183
8.5	Procédures et tableaux	184
8.5.1	Passage d'un tableau 1D en fortran	185
8.5.2	Passage d'un tableau 2D en fortran	187
8.5.3	Passage d'un tableau 1D en C	189
8.5.4	Passage d'un tableau 2D en C (norme C89)	191
8.5.5	Passage d'un tableau 2D automatique de dimensions variables en C99	195
8.5.6	Tableaux automatiques locaux : C99 et fortran	198

9	Allocation dynamique	199
9.1	Introduction	199
9.1.1	Trois types de tableaux	199
9.1.2	Cycle élémentaire d'un tableau dynamique	200
9.1.3	Allocation dynamique en C avec <code>malloc</code> ou <code>calloc</code>	202
9.1.4	Libération de la mémoire allouée en C avec <code>free</code>	204
9.2	Allocation d'un tableau 1D	205
9.2.1	Allocation d'un tableau 1D en fortran	205
9.2.2	Allocation d'un tableau 1D en C	206
9.3	Risques de fuite de mémoire	208
9.3.1	Fuite de mémoire avec les pointeurs en fortran	208
9.3.2	Fuite de mémoire en C	209
9.4	Application : manipulation de matrices	210

9.4.1	Matrices de taille quelconque en fortran	210
9.4.2	Matrices de taille quelconque en C : allocation en bloc	213
9.5	La bibliothèque <code>libmnitab</code>	220
9.6	Allocation dynamique en fortran 2003	221
10	Chaînes de caractères	225
10.1	Introduction	225
10.2	Déclaration, affectation des chaînes de caractères	226
10.3	Manipulation des chaînes de caractères	227
10.4	Chaînes de caractères en C	229
10.4.1	Déclaration de chaînes de longueur fixe en C	230
10.4.2	Exemple de chaînes de caractères de taille variable en C	233
10.4.3	Longueur d'une chaîne avec <code>strlen</code> et opérateur <code>sizeof</code>	236

10.4.4	Concaténation de chaînes avec <code>strcat</code>	237
10.4.5	Caractères non-ascii en C99 : caractères étendus et multi-octets	241
10.5	Chaînes de caractères en fortran	242
10.5.1	Sous-chaînes en fortran	242
10.5.2	Fonctions manipulant des chaînes en fortran	242
10.5.3	Tableaux de chaînes en fortran	243
10.5.4	Entrées-sorties de chaînes en fortran	244
10.5.5	Allocation dynamique de chaînes en fortran 2003	244
10.5.6	Passage de chaînes en argument en fortran	245
11	Entrées–sorties	246
11.1	Type de fichiers et accès : avantages respectifs	246
11.2	Entrées-sorties binaires (ou non formatées)	248

11.3 Entrées-sorties formatées (fichiers codant du texte)	249
11.4 Formats d'entrée–sortie	252
11.5 Exemple de lecture de fichier formaté en C	254
11.6 Exemple d'écriture de tableau 1D en fortran	256
12 Structures ou types dérivés	259
12.1 Intérêt des structures	259
12.2 Définition, déclaration et initialisation des structures	261
12.2.1 Définition de structures/types dérivés	261
12.2.2 Déclaration et initialisation d'objets de type structure	262
12.2.3 Contraintes d'alignement dans les structures en C	263
12.2.4 Implémentation des types dérivés en fortran	264
12.3 Manipulation des structures	265

12.4 Structures et tableaux	268
12.5 Structures/types dérivés, pointeurs et procédures	269
12.6 Exemple de structures auto-référencées : listes chaînées	274
12.7 Structures à composantes dynamiques	279
12.8 Conclusion sur les structures	281
13 Éléments de compilation séparée	282
13.1 Intérêt de la compilation séparée	282
13.2 Mise en œuvre robuste de la compilation séparée	282
13.3 Fichiers d'entête (<i>header files</i>) en C	287
13.3.1 Définition et usage	287
13.3.2 Structure d'un fichier d'entête	288
13.4 Exemple de programme C en plusieurs fichiers	289

13.5 Bibliothèques de fichiers objets	292
13.5.1 Bibliothèques statiques et bibliothèques dynamiques	292
13.5.2 Erreurs courantes lors de l'usage des bibliothèques	293
13.6 Commande de gestion des bibliothèques statiques : <code>ar</code>	294
13.6.1 Création d'une bibliothèque statique (archive)	295
13.6.2 Utilisation d'une bibliothèque statique en C (entête et archive)	296
13.6.3 Retour sur la bibliothèque standard du C	297
13.6.4 Retour sur la bibliothèque <code>libmnitab</code>	298
13.6.5 Bilan sur la création et l'usage d'une bibliothèque statique	299
13.7 Génération d'un fichier exécutable avec <code>make</code>	300
13.7.1 Principe	300
13.7.2 Construction d'un <code>makefile</code>	301
13.7.3 Exemple élémentaire de <code>makefile</code> en C	302
13.7.4 Utilisation d'un <code>makefile</code>	304

1 Introduction

1.1 Programmation en langage compilé

Conception, écriture et exécution d'instructions destinées à être traitées de manière automatique par un ordinateur.

Étapes (itérer si nécessaire) :

- **conception** : définir l'objectif du programme et la méthode à utiliser
⇒ algorithme puis organigramme puis pseudo-code
- **codage** : écrire le programme suivant la syntaxe d'un langage de haut niveau, portable et utilisant des bibliothèques : C, fortran, ...
⇒ code **source** : fichier texte avec instructions commentées
compréhensible pour le concepteur... et les autres
- **compilation** : transformer le code source en un code machine
⇒ code **objet** puis code **exécutable** : fichiers binaires compréhensibles par la machine
- **exécution** : tester le bon fonctionnement du programme
⇒ exploitation du code et production des résultats

Langage compilé et langage interprété

Langage compilé	Langage interprété
C , C++, fortran	shell , perl, php, scilab
<p>Le compilateur analyse l'ensemble du programme avant de le traduire en langage machine une fois pour toutes.</p> <p>⇒ optimisation possible</p> <p>L'exécutable produit est autonome mais non portable</p>	<p>L'interpréteur exécute le code source (script) instruction par instruction.</p> <p>⇒ pas d'optimisation</p> <p>L'interpréteur est nécessaire à chaque exécution.</p>
<p>En cas d'erreur de syntaxe à la dernière ligne du source le binaire n'est pas produit, donc pas d'exécutable !</p>	<p>seule la dernière instruction n'est pas exécutée !</p>
<p align="center">Compilé + interprété, par exemple java</p> <p>compilation → bytecode portable interprété par JVM (machine virtuelle java)</p>	

- L'ordinateur est muni d'un **système d'exploitation** (*exemple* : linux)
- Le **code source** est écrit dans un fichier texte au moyen d'un **éditeur de texte**.
Exemples : `vi`, `emacs`, `gedit`, `kate`, `kwite`, ... sous linux.
- Le **code machine (fichier objet)** généré par un **compilateur** :
programme qui analyse le code source, signale les erreurs de syntaxe, produit des avertissements sur les constructions suspectes, convertit un code source (sans erreur !) en code machine, optimise le code machine.

Exemples :

	C	fortran
<i>GNU Compiler Collection</i>	gcc	gfortran ou g95
Intel	<code>icc</code>	<code>ifort</code>
IBM	<code>xlc</code>	<code>xlf</code>

- L'éditeur de liens (**ld**) résout les appels aux bibliothèques → fichier **exécutable**
- Les instructions du programme sont exécutées par un **processeur** caractérisé par son architecture, la taille de ses registres (nombre de bits traités ensemble : 32, 64 bits), sa vitesse d'horloge, son jeu d'instructions...
Exemples : famille x86 d'Intel (32 bits), Pentium (32/64 bits) ou x64 (64 bits)

1.2 Historique

1.2.1 Langage fortran : Fortran = **F**ormula **T**ranslation

— 1954 : premier langage de **calcul scientifique** (télétypes, puis cartes perforées)

— ...

— 1978 : fortran V ou fortran 77

— 1991 : **fortran 90** (évolution majeure mais un peu tardive)

format libre, fonctions tableaux, allocation dynamique, structures, modules...

⇒ **ne plus écrire de fortran 77**

— 1997 : fortran 95 = mise à jour mineure

— 2004 : **fortran 2003** standard adopté

nouveautés : interopérabilité avec C, arithmétique IEEE, accès au système,

allocations dynamiques étendues à d'autres contextes, aspects objet...

fortran 2003 implémenté sur certains compilateurs (en cours pour `gfortran`)

<http://fortranwiki.org/fortran/show/Fortran+2003+status>

— 2010 : standard **fortran 2008** adopté le 20 sept 2010

<http://fortranwiki.org/fortran/show/Fortran+2008+status>

— futur standard fortran 2015 en cours de rédaction

1.2.2 Langage C

- langage conçu dans les années 1970
- 1978 : **The C Programming Language** de B. KERNIGHAN et D. RICHIE
- développement lié à la diffusion du système UNIX
- 1988–90 : normalisation **C89** ANSI–ISO (bibliothèque standard du C)
Deuxième édition du KERNIGHAN et RICHIE **norme ANSI**
- 1999 : norme **C99**, implémentation <http://gcc.gnu.org/c99status.html>
nouveaux types (booléen, complexe, entiers de diverses tailles (prise en compte des processeurs 64 bits), caractères larges (unicode), ...),
introduction de la généricité dans les fonctions numériques,
déclarations tardives des variables, tableaux automatiques de taille variable...
⇒ se conformer à une **norme** pour la portabilité
- norme **C11** (ex-C1x) parue en avril 2011 (généricité via cpp, unicode, ...)
- base d'autres langages dont le **C++** (premier standard en 1998)
puis java, php, ...

1.3 Intérêts respectifs

Langage fortran	Langage C
langage de haut niveau (structures de contrôle, structures de données, fonctions, compilation séparée, ...)	
tableaux multidimensionnels et fonctions associées (cf. matlab et scilab)	
	mais aussi ... langage de bas niveau (manipulation de bits, d'adresses, ...)
applications scientifiques et de gestion	
langage portable grâce à la norme et à des bibliothèques	
langage puissant, efficace, mais aussi ... peu permissif !	langage puissant, efficace, mais aussi ... permissif !
	écriture de systèmes d'exploitation

⇒ **Présentation comparative** succincte

pour un public connaissant déjà au moins un langage compilé.

Exemple : appel à une procédure de la bibliothèque LAPACK pour le calcul des éléments propres d'une matrice $m \times m$ réelle.

Langage C

```
info = lapacke_sgeev(CblasColMajor, 'V', 'N',  
    m, (float *)matrice,  
    m, (float *)valpr_r, (float *)valpr_i,  
    (float *)vectpr_l, m, (float *)vectpr_r, m);
```

12 arguments + status de retour optionnel

Langage Fortran 90

```
call la_ggev(matrice, valpr_r, valpr_i, &  
    vectpr_l, vectpr_r, info= info)
```

5 arguments seulement + status optionnel passé par **mot-clef**

1.4 Généralités

langage C		langage Fortran
format « libre »	format du code	format libre du fortran 90 (format fixe en fortran 77)
⇒ mettre en évidence les structures par la mise en page (indentation)		
pas une entité particulière (la fin de ligne est un séparateur comme l'espace, la tabulation, ...)	ligne	la fin de ligne termine une instruction & en fin de ligne permet de continuer une instruction sur la ligne suivante
instructions simples terminées par ;	;	; pour séparer plusieurs instructions sur une même ligne
entre /* et */ (peut s'étendre sur plusieurs lignes) (pas d'imbrication selon la norme) en C99 et C++ introduit par // et terminé en fin de ligne	commentaire	introduit par ! et terminé en fin de ligne
identificateur : au plus 31 caractères alphanumériques plus _ , commençant par une lettre peut commencer par _		au plus 63 caractères en f2003
distinction	maj/minuscule	pas de distinction
lignes de directives pour le préprocesseur : # en première colonne		

1.5 Exemple de programme C avec une seule fonction utilisateur : main

```
#include <stdio.h>          /* instructions préprocesseur */
#include <stdlib.h>         /* pas de ";" en fin          */
int main(void)             /* fonction principale        */
{                           /* <<= début du corps        */
    int i ;                /*          déclarations     */
    int s=0 ;              /*          initialisation    */
    for (i = 1 ; i <= 5 ; i++)
    {                       /* <<= début de bloc         */
        s += i ;
    }                       /* <<= fin de bloc           */
    printf("somme des entiers de 1 à 5\n") ;
    printf("somme = %d\n", s) ; /*          affichage        */
    exit(0) ;              /* renvoie à unix le status 0 */
}                           /* <<= fin du corps         */
```

1.6 Exemple de programme fortran avec une seule procédure

```
program ppal                                ! << début du programme ppal
  implicit none                             ! nécessaire en fortran
  integer :: i                               ! déclarations
  integer :: s = 0                          ! initialisation
  do i = 1, 5                                ! structure de boucle
    s = s + i
  end do                                     ! <<= fin de bloc
  write(*,*) "somme des entiers de 1 à 5"
  write(*,*) "somme = ", s                 ! affichage
  stop
end program ppal                            ! << fin du programme ppal
```

1.7 Structure générale d'un programme C élémentaire

```
#include <stdio.h>

#include "fonction.h"

int main(void) {

    Déclarations des variables

    Instructions exécutables

}
```

Structure élémentaire d'un programme C89

- Une **instruction simple** doit se terminer par **;**
- Une **instruction composée**, délimitée par des accolades **{** et **}** est constituée d'un **bloc d'instructions** simples ou composées (imbrication possible...)
- La définition d'une **fonction** se compose d'un **entête** et d'un **corps** (entre **{** et **}**) qui est en fait une instruction composée.
- L'**entête** d'une fonction spécifie le **type** de la valeur de retour, le nom de la fonction et ses paramètres ou arguments avec leurs types :
type valeur_de_retour (type1 arg1, type2 arg2, ...)
où chaque argument est déclaré par son type, suivi de son identificateur
- Un **programme C** = une ou plusieurs **fonctions** dont au moins la fonction **main** : le programme principal.
N.-B : à l'extérieur de ces fonctions, il peut comporter des déclarations de variables globales, des déclarations de fonctions spécifiant leur prototype (interface en fortran), et des directives pour le préprocesseur introduites par **#**.

1.8 Exemple de programme C : 2 fonctions (main et somme)

```
1 /* programme elem.c */
2 /* instructions préprocesseur */
3 #include <stdio.h> /* pour les entrées/sorties */
4 #include <stdlib.h> /* par exemple pour exit */
5 /* fin des instructions préprocesseur */
6 int somme(const int p) ; /* déclaration de la fonction somme */
7 int main(void) /* fonction principale (sans param.) */
8 { /* <=< début de bloc */
9     int s ; /* déclaration de l'entier s */
10
11     printf("somme des entiers de 1 à 5\n"); /* "\n" => retour ligne */
12     s = somme(5) ; /* appel de la fonction somme */
13     printf("somme = %d\n", s) ; /* impression du résultat */
14     exit(0) ; /* renvoie à unix un status 0 (OK) */
15 }
```

Suite du code : la fonction **somme**

```
16 /*                                                    */
17 int somme(const int p)      /* définition de la fonction somme */
18 /* calcul de la somme des p premiers entiers */
19 {                          /* <<= début du corps de la fct    */
20     int i;                  /* déclaration des var. locales */
21     int sum = 0;            /* et initialisation de sum     */
22     for (i = 0; i <= p ; i++) /* structure de boucle         */
23     {                        /* <<= début de bloc          */
24         sum += i ;           /* sum = sum + i               */
25         printf(" i = %d, somme partielle = %d\n", i, sum) ;
26     }                        /* <<= fin de bloc            */
27     return sum ;           /* valeur rendue par la fonction */
28 }                          /* <<= fin du corps de la fct    */
```

1.9 Structure générale d'un programme fortran élémentaire

```
PROGRAM ppal  
  
USE (des modules éventuels)  
  
IMPLICIT NONE  
  
Déclarations des variables  
  
Instructions exécutables  
  
END PROGRAM ppal
```

Structure élémentaire d'un programme fortran

- Un **programme** fortran = un programme principal

program *identificateur*

...

end program *identificateur*

plus éventuellement des **procédures** :

- **fonctions** avec valeur de retour typée :

[*type*] **function** *identificateur*(*arg1*, *arg2*, ...)

...

end function *identificateur*

appelées quand elles apparaissent dans une expression

- **sous-programmes** :

subroutine *identificateur*(*arg1*, *arg2*, ...)

...

end subroutine *identificateur*

appel explicite par

call *identificateur*(*arg1*, *arg2*, ...)

- Les **procédures** peuvent être :
 1. **internes** à un programme ou une autre procédure : introduites par **contains** et non réutilisables
 2. **externes** : (comme en fortran 77)
 - N.B. : une procédure peut être externe mais dans le même fichier que l'appelant
 - ⇒ compilation séparée sans contrôle interprocédural
 - ⇒ fournir l'interface à l'appelant pour permettre ce contrôle
 - ⇒ duplication de code comme en C ... risque d'incohérences
 3. **intégrées dans des modules** externes (après **contains**)
 - ⇒ l'interface est mémorisée dans un fichier de module lors de la compilation
 - ⇒ **use** `nom_module` permet à l'appelant de connaître l'interface en relisant le fichier de module
 - ⇒ **solution la plus portable**
- à l'extérieur des programmes ou procédures, pas de déclarations de variables sauf variables globales dans un module

1.10 Ex. de programme fortran avec fonction dans un module

```

1  module m_som                                ! << début du module m_som
2  contains                                    !   qui héberge des procédures
3  function somme(p)                             !   << début de la fonction somme
4  implicit none                               ! nécessaire en fortran
5  integer :: somme                             ! type du résultat
6  integer, intent(in) :: p                   ! argument d'entrée non modifiable
7  integer :: i, n                             ! déclaration des variables locales
8  n = 0                                        ! initialisation
9  do i = 0, p                                 ! structure de boucle avec compteur
10     n = n + i                               !   cumul
11     write(*,*) " i = ", i, " somme partielle = ", n
12 end do                                       !   <=< fin de bloc
13     somme = n                                ! affectation de la valeur de retour
14     return                                  ! retour à l'appelant
15 end function somme                             !   << fin de la fonction somme
16 end module m_som                             ! << fin du module m_som

```

Suite du code : programme principal

```
17 !-----  
18 program ppal           ! << début du programme ppal  
19 use m_som             ! lit m_som.mod => fournit interface  
20 implicit none         ! nécessaire en fortran  
21 integer :: s          ! déclaration de l'entier s  
22 write(*,*) "somme des entiers de 1 à 5" ! retour ligne par défaut  
23 s = somme(5)           ! appel de la fonction somme  
24 write(*,*) "somme = ", s ! impression du résultat  
25 stop                  ! ne renvoie pas de status  
26 end program ppal      ! << fin du programme ppal  
27 !-----
```

1.11 Compilation et édition de liens

- Fichier **source** (texte) de suffixe **.c** en C (**.f90** en fortran 90)
- Fichier **objet** (binaire) de suffixe **.o**
- Fichier **exécutable** (binaire) **a.out** par défaut

La commande de compilation `gcc toto.c gfortran/g95 toto.f90`

lance par défaut trois actions :

1. traitement par le **préprocesseur** (**cpp**) des lignes commençant par **#**
appelées directives (transformation textuelle)
2. **compilation** à proprement parler → fichier objet **toto.o**
3. **édition de liens** (*link*) : le compilateur lance **ld** → fichier exécutable **a.out**
assemblage des codes objets et résolution des appels aux bibliothèques

N.-B. : seul le fichier source est portable (indépendant de la machine)

(1)

édition | vi essai.f90 / emacs essai.f90

fichier source
essai.f90

gfortran essai.f90 -o essai.x

(2)

(préprocesseur)

compilation | gfortran -c essai.f90

fichier objet
essai.o

compilation + lien
(2) + (3)

(3)

lien | gfortran essai.o -o essai.x

fichier exécutable
essai.x

(4)

exécution | essai.x

Principales options de compilation

Options de compilation permettant de **choisir les étapes et les fichiers** :

- `gcc -E toto.c` : préprocesseur seulement
- `-c` : préprocesseur et compilation seulement
- `-o toto.x` : permet de spécifier le nom du fichier exécutable
- `-ltruc` donne à `ld` l'accès à la **bibliothèque `libtruc.a`**
ex. : `-lm` pour `libm.a`, bibliothèque mathématique indispensable en C
option à placer **après** les fichiers appelants

Options de compilation **utiles à la mise au point** :

- vérification des standards du langage : `-std=c99` ou `-std=f95`
 - avertissements (*warnings*) sur les instructions suspectes (variables non utilisées, instructions apparemment inutiles, changement de type, ...) : `-Wall`
 - vérification des passages de paramètres
(nécessite un contrôle interprocédural, donc les prototypes ou interfaces)
- ⇒ **faire du compilateur un assistant efficace** pour anticiper les problèmes avant des erreurs à l'édition de liens ou, pire, à l'exécution.

1.12 Compilateurs utilisés

langage C	fortran
linux	
gcc	g95 gfortran
avec options sévères et précision de la norme	
alias gcc-mni-c89	alias g95-mni ou gfortran-mni
C99 → alias gcc-mni-c99	f2003 → alias g2003-mni ou gfortran2003-mni
http://gcc.gnu.org	http://www.g95.org http://gcc.gnu.org/fortran/

2 Types et déclarations des variables

Langages **fortement typés** => déclarer le type de chaque variable

2.1 Types de base

Types représentés **exactement**

langage C	Type	fortran 90
<code>void</code>	vide	<i>////</i>
<i>////</i> C99 <code>bool</code>	booléen	logical
<code>char</code>	caractère	\approx <code>character(len=1)</code>
C99 <code>wchar_t</code>	caractère large unicode	\approx <code>character(kind=ucs4, len=1)</code>
<i>////</i> (tableau de <code>char</code>)	chaîne de caractères	character(len=...)
<code>short int</code>	entier court	<code>integer(kind=2)</code>
int	entier	integer
<code>long int</code>	entier long	<code>integer(kind=4/8)</code>
C99 <code>long long</code>	entier long long	<code>integer(kind=8)</code>

Types représentés **approximativement**
en virgule flottante : mantisse et exposant

langage C	Type	fortran 90
float	réel simple précision	real
double	double précision	double precision
long double	précision étendue	real(kind=10/16)
////C99 complex #include <tgmath>	complexe + variantes	complex

2.2 Domaine et précision

2.2.1 Domaine des entiers signés

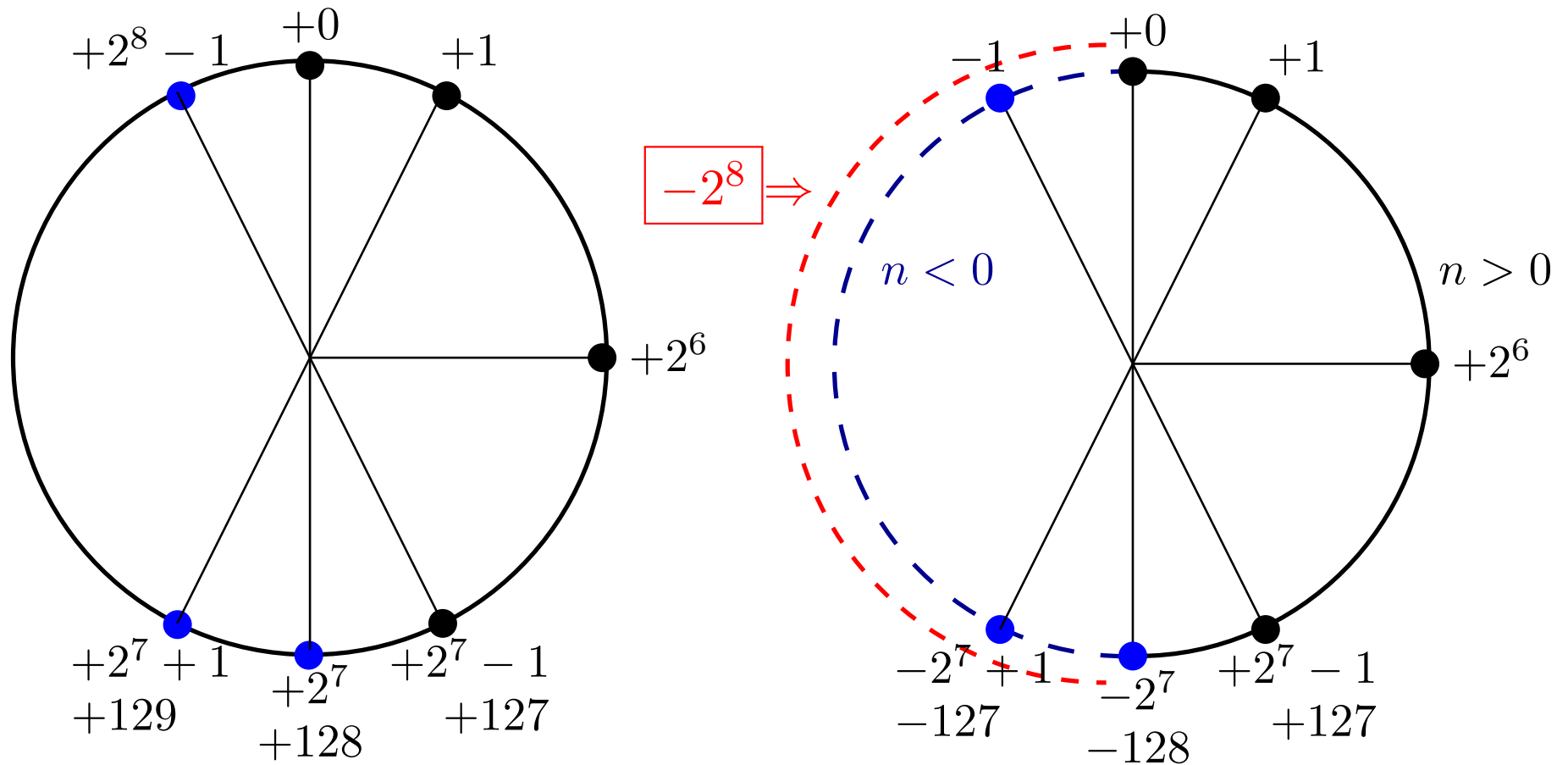
Langage C : Attributs **unsigned** et **signed** des types entiers ou caractères pour indiquer si le poids fort est un **bit de signe** (cas par défaut pour les entiers).

Taille et domaine des types de base (dépend de la machine et du compilateur)

type	taille	unsigned	signed
caractère	1 octet	$0 \rightarrow 255 = 2^8 - 1$	$-128 \rightarrow +127$
entier court	2 octets	$0 \rightarrow 2^{16} - 1$	$-2^{15} \rightarrow 2^{15} - 1$
entier	4 octets	$0 \rightarrow 2^{32} - 1$	$-2^{31} \rightarrow 2^{31} - 1$
entier long	$(4^a \text{ ou } 8^b \text{ octets})$	$0 \rightarrow 2^{64} - 1$	$-2^{63} \rightarrow 2^{63} - 1$

a. Machine 32 bits

b. Machine 64 bits



Entiers **positifs** sur 8 bits

Discontinuité en haut à gauche de 0

Entiers **relatifs** sur 8 bits

Discontinuité en bas entre $+127$ et -128

Pour passer des positifs aux relatifs, on soustrait 2^8 dans la partie gauche du cercle.

Rappel : $\log_{10} 2 \approx 0,30 \Rightarrow 2^{10} = 1024 = 10^{10 \log_{10}(2)} \approx 10^3$

	fortran	C	
sur 32 bits = 4 octets	HUGE (1)	INT_MAX	$2^{31} \approx 2 \times 10^9$
sur 64 bits = 8 octets	HUGE (1_8)	LLONG_MAX	$2^{63} \approx 9 \times 10^{18}$

⇒ C99 : types entiers étendus à nb d'octets imposé ou à minimum imposé

par exemple : `int32_t` ou `int_least64_t`

⇒ en fortran, choix des variantes (**KIND**) de types selon le domaine (*range*) par

SELECTED_INT_KIND pour les entiers.

Dépassement de capacité en entier ⇒ passage en négatif

2.2.2 Valeurs maximales des entiers en C

Définies dans `/usr/include/limits.h`

(et `/usr/include/stdint.h` en C99)

Le code d'impression C99 par curiosité :

```
#include<stdio.h>                // limites-int-machine-c99.c
#include<stdlib.h>
#include<limits.h>
#include<stdint.h>
int main(void) {
    // impression des valeurs limites des entiers sur la machine
    // non-signés puis signés en décimal, hexadécimal et octal
    // version C99
    /* entier long long */ // C99 seulement
    printf("%-18s %20llu %16llx %22llo\n",
           "Unsig-Lng-Lng-max", UINTMAX_MAX, UINTMAX_MAX, UINTMAX_MAX);
    printf("%-18s %20lld %16llx %22llo\n",
           "Long-Long-max", LLONG_MAX, LLONG_MAX, LLONG_MAX);
```

```
/* entier long */
printf("%-18s %20lu %16lx %22lo\n",
    "Unsigned-Long-max", ULONG_MAX, ULONG_MAX, ULONG_MAX);
printf("%-18s %20ld %16lx % 22lo\n",
    "Long-max", LONG_MAX, LONG_MAX, LONG_MAX);
/* entier */
printf("%-18s %20u %16x % 22o\n",
    "Unsigned-Int-max", UINT_MAX, UINT_MAX, UINT_MAX);
printf("%-18s %20d %16x % 22o\n",
    "Int-max", INT_MAX, INT_MAX, INT_MAX);
/* entier court */
printf("%-18s %20hu %16hx % 22ho\n",
    "Unsigned-Short-max", USHRT_MAX, USHRT_MAX, USHRT_MAX);
printf("%-18s %20hd %16hx % 22ho\n",
    "Short-max", SHRT_MAX, SHRT_MAX, SHRT_MAX);
exit(0) ;
}
```

Valeurs maximales des entiers en C : machine 32 bits

```
Unsig-Lng-Lng-max 18446744073709551615 ffffffffffffffff 17777777777777777777
Long-Long-max     9223372036854775807 7fffffffffffffff 77777777777777777777
```

Unsigned-Long-max	4294967295	ffffffff	3777777777
Long-max	2147483647	7fffffff	1777777777
Unsigned-Int-max	4294967295	ffffffff	3777777777
Int-max	2147483647	7fffffff	1777777777
Unsigned-Short-max	65535	ffff	17777
Short-max	32767	7fff	7777

Valeurs maximales des entiers en C : machine 64 bits

```
Unsig-Lng-Lng-max 18446744073709551615 ffffffffffffffff 17777777777777777777
Long-Long-max     9223372036854775807 7fffffffffffffff 77777777777777777777
```

Unsigned-Long-max	18446744073709551615	fffffffffffffff	17777777777777777777
Long-max	9223372036854775807	7fffffffffffffff	77777777777777777777
Unsigned-Int-max	4294967295	ffffffff	3777777777
Int-max	2147483647	7fffffff	1777777777
Unsigned-Short-max	65535	ffff	17777
Short-max	32767	7fff	7777

2.2.3 Domaine et précision des flottants

Par exemple **en base 10**, avec 4 chiffres après la virgule, on peut comparer les deux représentations approchées (par troncature et non par arrondi, pour simplifier) :

nombre exact	virgule fixe	virgule flottante
	par troncature	
0.000123456789	.0001	0.1234×10^{-3}
0.00123456789	.0012	0.1234×10^{-2}
0.0123456789	.0123	0.1234×10^{-1}
0.123456789	.1234	0.1234×10^0
1.23456789	1.2345	0.1234×10^1
12.3456789	12.3456	0.1234×10^2
123.456789	123.4567	0.1234×10^3
1234.56789	1234.5678	0.1234×10^4

Représentation approchée en **virgule flottante** pour concilier dynamique et précision

En binaire, nombre de bits réparti entre **mantisse** (partie fractionnaire) et **exposant**

- nombre de bits de l' **exposant** \Rightarrow **domaine fini**
- nombre de bits de la **mantisse** \Rightarrow **précision limitée**

ε = la plus petite valeur telle que $1 + \varepsilon$ soit le successeur de 1

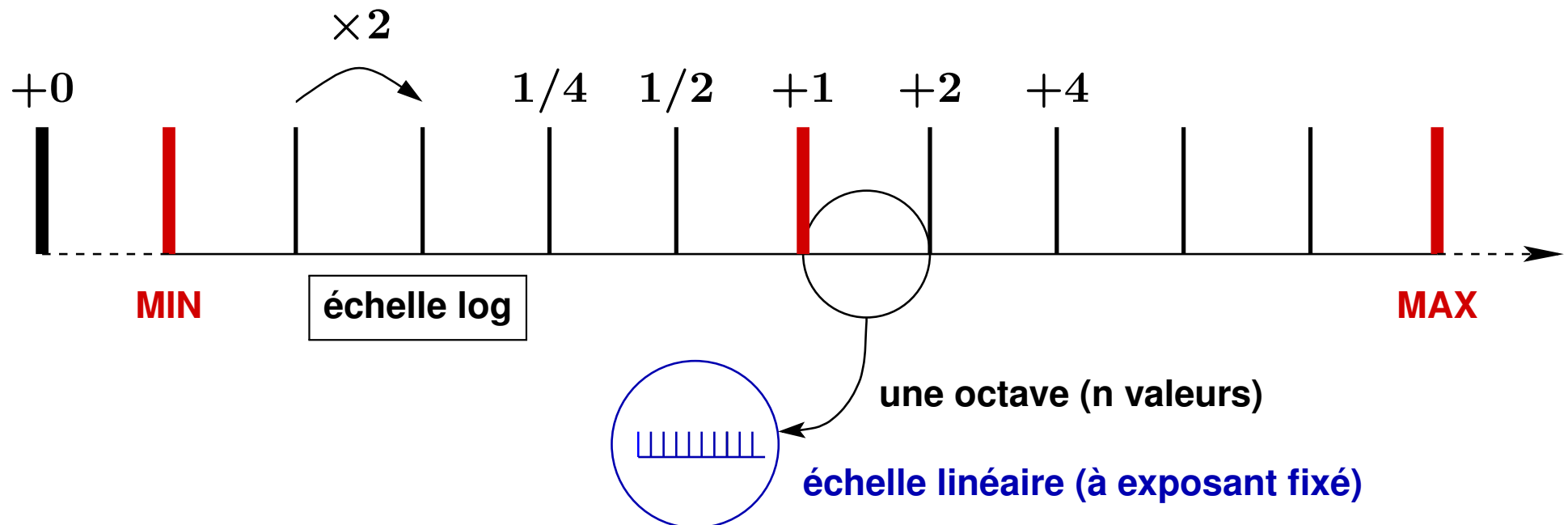
ε est le pas des flottants dans l'octave $[1, 2[$.

	fortran	C	
simple préc.	HUGE (1.)	FLT_MAX	$3,4 \times 10^{38}$
= 4 octets	TINY (1.)	FLT_MIN	$1,18 \times 10^{-38}$
= 32 bits	EPSILON (1.)	FLT_EPSILON	$2^{-23} \approx 1,2 \times 10^{-7}$
double préc.	HUGE (1.d0)	DBL_MAX	$1,8 \times 10^{308}$
= 8 octets	TINY (1d0)	DBL_MIN	$2,2 \times 10^{-308}$
= 64 bits	EPSILON (1.d0)	DBL_EPSILON	$2^{-52} \approx 2,2 \times 10^{-16}$

Domaine des flottants fixé par le nombre de bits q de l'exposant

En virgule flottante, les octaves sont en **progression géométrique de raison 2** :

Nombre d'octaves $\approx 2^q$, réparties presque symétriquement autour de 1.



Domaine des flottants positifs normalisés (**échelle log**)

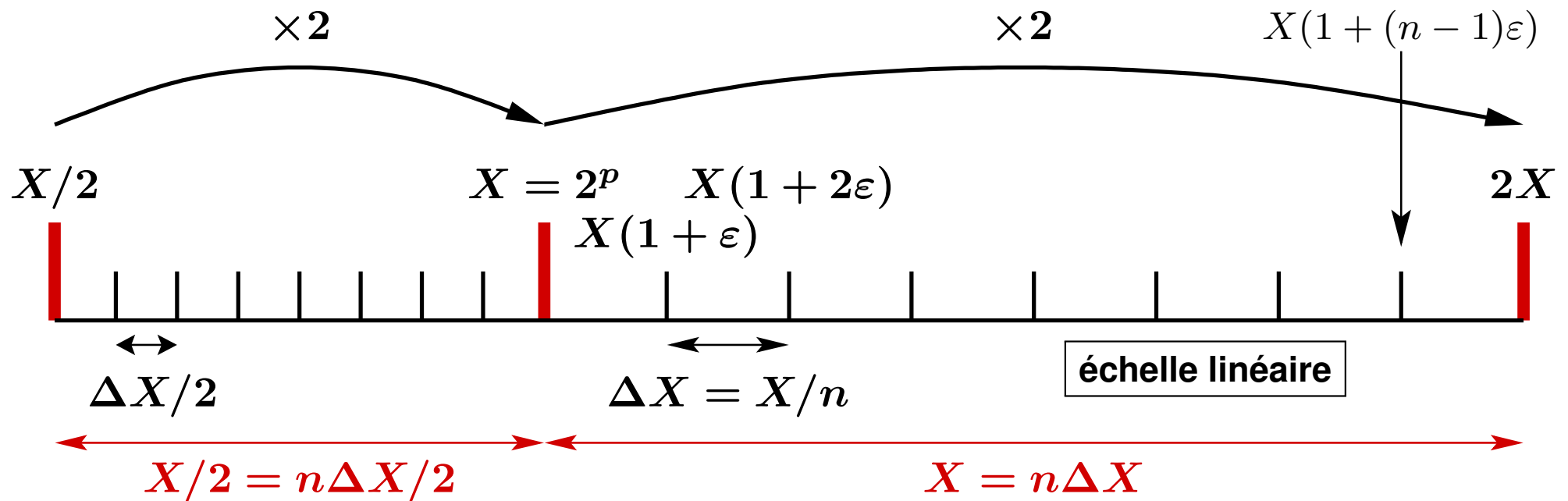
Flottants sur 32 bits : **8 bits d'exposant** donc $254 \approx 2^8$ octaves

$$\text{MAX} \approx 1/\text{MIN} \approx 2^{127} \approx 1,7 \times 10^{38}$$

Précision des flottants fixée par le nombre de bits m de la mantisse

Dans chaque octave $[2^p, 2^{p+1}[= [X, 2X[$, l'exposant est constant

$\Rightarrow n = 2^m$ flottants en **progression arithmétique** de pas $\Delta X = X/n = \varepsilon X$



Exemple représenté ici : deux octaves de flottants positifs avec mantisse sur $m=3$ bits

$n = 2^m = 2^3 = 8$ intervalles et aussi 8 valeurs par octave

Sur 32 bits : $m=$ **23 bits de mantisse** $\Rightarrow \varepsilon = \Delta X/X = 1/n = 2^{-23} \approx 10^{-6}/8$

2.2.4 Caractéristiques des types numériques en fortran

DIGITS (x)	nombre de bits de $ x $ si entier, de sa mantisse si réel
PRECISION (x)	nombre de chiffres (décimaux) significatifs de x
EPSILON (x)	plus grande valeur du type de x négligeable devant 1
RANGE (x)	puissance de 10 du domaine de x (plus petite valeur absolue)
TINY (x)	plus petite valeur positive représentable dans le type de x
HUGE (x)	plus grande valeur positive représentable dans le type de x

Portabilité numérique du code

⇒ demander la variante (**KIND**) du type numérique suffisante

k_i = **SELECTED_INT_KIND** (r) pour les entiers allant jusqu'à 10^r

k_r = **SELECTED_REAL_KIND** (p, r) pour les réels

pour un domaine de 10^{-r} à 10^r et une précision 10^{-p} (p chiffres significatifs)

Entiers par défaut

compil/proc.	option	KIND	DIGITS	HUGE
g95/32 bits	-i4	4	31	$2147483647 = 2^{31} - 1$
g95/64 bits	-i8	8	63	$9223372036854775807 = 2^{63} - 1$

Réels par défaut

option g95	nb. octets	Digits	Préc.	EPSILON	TINY	HUGE
-r4	4	24	7	1.192093E-7	1.175494E-38	3.402823E+38
-r8	8	53	15	2.220 E-016	2.225 E-308	1.79769 E+308

2.3 Déclarations des variables

Déclarer une variable = réserver une zone en mémoire pour la stocker

Variable typée \Rightarrow lui associer un nombre de bits et un codage

(transformation entre valeur et état des bits en mémoire vive)

La taille de la zone et le codage dépendent du type de la variable.

Les bits de la zone ont au départ des valeurs imprévisibles, sauf si...

Initialiser une variable = affecter une valeur à une variable

au moment de la réservation de la mémoire

Déclarer **en tête des procédures** : **obligatoire en fortran** et conseillé en C89

En **C99** : déclarations tardives autorisées mais préférer en tête de bloc

langage C \geq 89	fortran \geq 90
en tête des blocs C99 N'importe où	en tête des procédures
Syntaxe	
type identificateur1, identificateur2 ... ;	type :: identificateur1, identificateur2, ...
Exemple : déclaration de 3 entiers	
int i, j2, k_max ;	integer :: i, j2, k_max
Initialisation : lors de la déclaration	
int i = 2 ; (exécution)	integer :: i = 2 (compilation)
Déclaration de constantes (non modifiables)	
const int i = 2 ; penser aussi à #define VAR 2	integer, parameter :: i = 2 utilisable comme une vraie constante



2.4 Exemples d'attribut constante

2.4.1 Exemples d'attribut const en C

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  const int i = 2 ; /* non modifiable */
  /* int i = 2 ; */ /* serait modifiable */
  i++ ; /* erreur (gcc>v4.0) à la compilation */
  printf("i vaut %d\n", i);
  exit(0);
}
```

Avec gcc par exemple

```
const.c:9: error: increment of read-only variable 'i'
```


2.4.2 Exemples d'attribut parameter en fortran

```
program constante
implicit none
integer, parameter :: i = 2 ! non modifiable
! integer :: i = 2 ! serait modifiable
i = i + 1 ! => erreur à la compilation
write(*,*) "i vaut ", i
end program constante
```

Avec g95 par exemple

```
i = i + 1 ! => erreur à la compilation
1
```

Error: Expected 'i' at (1) to be a VARIABLE

2.5 Les constantes

langage C	C99	Type	fortran 90
<code>//// C99 : true ($\neq 0$) false (0)</code>		booléen	<code>.TRUE.</code> <code>.FALSE.</code>
<code>' a '</code>		caractère	<code>' a '</code> <code>" a "</code>
<code>"chaine"</code>	<code>"s'il"</code>	chaînes	<code>' chaine '</code> <code>" s' il "</code>
17		entier court	17_2
17		entier décimal	17
021 (attention)		entier 17 ₁₀ en octal	O' 21'
0x11		entier en hexadécimal	Z' 11'
17 L		entier long	17_8
-47.1 f	-6.2e-2 f	réel simple précision	-47.1 -6.2e-2
-47.1	-6.2e-2	double précision	47.1_8 -6.2e-2_8
-47.1 L	-6.2e-2 L	double précision long	47.1_16 -6.2e-2_16
<code>////</code> sauf I C99 :	<code>2.3-I*.5</code>	complexe	<code>(2.3, -5.)</code>



3 Opérateurs

3.1 Opérateur d'affectation

L'affectation `=` peut provoquer une **conversion de type implicite !**

⇒ problèmes de représentation des valeurs numériques :

- étendue (*range*) : ex. dépassement par conversion flottant vers entier
- précision : ex. conversion entier exact vers flottant approché

⇒ Préférer les conversions **explicites**

langage C	fortran
lvalue = (type) expression	variable = type (expression)
⇒ conversion forcée (opérateur cast)	grâce à des fonctions intrinsèques
<code>entier = (int) flottant;</code>	<code>entier = INT (flottant)</code>

```

#include <stdio.h>                                /* programme precision.c */
#include <stdlib.h>
int main(void) {
    int n = 123456789;        // exact
    float b = 0.123456789f;  // approché
    float nf;
    printf("float: %d octets \t int: %d octets\n",
           (int) sizeof(float), (int) sizeof(int));
    nf = (float) n;          // conversion => approché à 10(-7)
    printf("n (int)      = %d \nnf (float) = %.10g \n"
           "b (float) =%.10g\n", n, nf, b);
    exit(EXIT_SUCCESS);
}

```

float: 4 octets

int: 4 octets

n (int) = 123456789

exact

nf (float) = 123456792

approché

b (float) = 0.123456791

3.2 Opérateurs algébriques

	langage C	fortran 90	difficultés
addition	+	+	
soustraction	-	-	
multiplication	*	*	
division	/	/	div. entière
élévation à la puissance	≈ pow (x, y)	**	
reste modulo	%	≈ mod (i, j)	avec négatifs



Opérations binaires \Rightarrow même type pour les opérandes (sauf réel ****** entier fortran)

Types différents \Rightarrow **conversion implicite** vers le type le plus riche avant opération

3.3 Opérateurs de comparaison

	langage C	fortran 90
→ résultat	entier	booléen
inférieur à	<	<
inférieur ou égal à	<=	<=
égal à	==	==
supérieur ou égal à	>=	>=
supérieur à	>	>
différent de	!=	/=



mais = pour affectation

3.4 Opérateurs logiques

	langage C ^a	fortran 90 ^b
ET	&&	.AND.
OU	 	.OR.
NON	!	.NOT.
EQUIVALENCE	////	.EQV.
OU exclusif	////	.NEQV.

a. Rappel : pas de type booléen en C89 (faux=0, vrai si $\neq 0$), mais on peut le simuler avec une énumération : `enum boolean {faux = 0, vrai = 1};` mais le type booléen (`bool`) existe en C99, avec les constantes `true` et `false` définies dans `stdbool.h`.

b. Rappel : constantes booléennes (`.TRUE.` et `.FALSE.`) en fortran 90

3.5 Incrémentation et décrémentation en C

— **post-incrémentation** / **post-décrémentation**

i++ incrémente / **i--** décrémente **i** d'une unité,

après évaluation de l'expression

`p=2; n=p++;` donne `n=2` et `p=3`

`p=2; n=p--;` donne `n=2` et `p=1`


— **pré-incrémentation** / **pré-décrémentation**

++i incrémente / **--i** décrémente **i** d'une unité,

avant évaluation de l'expression

`p=2; n=++p;` donne `n=3` et `p=3`


`p=2; n=--p;` donne `n=1` et `p=1`

 **i = i++**; indéterminé ! \Rightarrow pas d'expression comportant `i++` affectée à `i`
mais **i = ++i**; équivaut à `++i`; ou à `i++`;

3.6 Opérateurs d'affectation composée en C

Ivalue opérateur = expression \Rightarrow Ivalue = Ivalue opérateur expression

Exemples :

	j	+=	i	\Rightarrow	j	=	j	+	i
	b	*=	$a + c$	\Rightarrow	b	=	b	*	(a + c)

3.7 Opérateur d'alternative en C

**exp1 ? exp2 : exp3 \Rightarrow si exp1 est vraie, exp2 (alors exp3 n'est pas évaluée)
sinon exp3 (alors exp2 n'est pas évaluée)**

Exemple :

$c = (a > b) \text{ ? } a \text{ : } b$ affecte le max de a et b à c

3.8 Opérateurs agissant sur les bits

Le langage C possède des opérateurs de bas niveau travaillant directement sur les champs de bits. Il existe, en fortran 90, des fonctions d'arguments entiers rendant des services similaires.

langage C	signification	fortran 90
\simexpr	négation	\approx NOT (expr)
expr1 & expr2	et	\approx IAND (expr1, expr2)
expr1 expr2	ou	\approx IOR (expr1, expr2)
expr1 ^ expr2	ou exclusif	\approx IEOR (expr1, expr2)
expr1 << expr2	décalage à gauche de expr2 bits	\approx ISHFT (expr1, expr2)
expr1 >> expr2	décalage à droite de expr2 bits	\approx ISHFT (expr1, -expr2)

3.9 Opérateur sizeof en C

Taille en octets d'un objet ou d'un type (résultat de type `size_t`).

Cet opérateur permet d'améliorer la portabilité des programmes.

sizeof identificateur

```
size_t n1; double a;  
n1 = sizeof a;
```

sizeof (type)

```
size_t n2;  
n2 = sizeof(int);
```

3.10 Opérateur séquentiel « , » en C

expr1 , expr2 permet d'évaluer successivement les expressions **expr1** et **expr2**.

Utilisé essentiellement dans les structures de contrôle (`if`, `for`, `while`).

3.11 Opérateurs & et * en C

&objet ⇒ adresse de l'objet

***pointeur** ⇒ objet pointé (indirection)

3.12 Priorités des opérateurs en C

- opérateurs unaires **+**, **-**, **++**, **--**, **!**, **~**, *****, **&**, **sizeof**, (cast)
- opérateurs algébriques *****, **/**, **%**
- opérateurs algébriques **+**, **-**
- opérateurs de décalage **<<**, **>>**
- opérateurs relationnels **<**, **<=**, **>**, **>=**
- opérateurs relationnels **==**, **!=**
- opérateurs sur les bits **&**, puis **^**, puis **|**
- opérateurs logiques **&&**, puis **||**
- opérateur conditionnel **?** **:**
- opérateurs d'affectation **=** et les affectations composées
- opérateur séquentiel **,**

⇒ indiquer les priorités avec des parenthèses !

4 Entrées et sorties standard élémentaires

C	fortran 90
écriture sur stdout = écran	
<pre>printf ("format", liste d'expressions);</pre>	<pre>WRITE (*, *) & liste d'expressions</pre>
lecture depuis stdin = clavier	
<pre>scanf ("format", liste de pointeurs);</pre>	<pre>READ (*, *) & liste de variables</pre>
format %d , %g ou %s ... selon le type pour chaque variable (gabarit optionnel)	format libre (*) le plus simple mais on peut préciser
spécifier \n en sortie pour changer de ligne	forcer par / dans le format changement d'enregistrement à chaque ordre READ ou WRITE

4.1 Introduction aux formats d'entrée–sortie

Correspondance très approximative entre C et fortran (**w** =largeur, **p**= précision)

	c	fortran 90
entiers		
décimal	<code>%w[.p]d</code>	<code>Iw [.p]</code>
	<code>%d</code>	<code>I0</code>
réels		
virgule fixe	<code>%w[.p]f</code>	<code>Fw .p</code>
virgule flottante	<code>%w[.p]e</code>	<code>Ew .p</code> <code>ESw .p</code> (scientif.) <code>ENw .p</code> (ingénieur)
préférer ⇒ général	<code>%w[.p]g</code>	<code>Gw .p</code>
caractères		
caractères	<code>%w[.p]c</code>	<code>A [w]</code>
chaîne	<code>%w[.p]s</code>	<code>A [w]</code>

```
#include <stdio.h> /* entrées sorties standard */
#include <stdlib.h>

int main(void) {
int i;
float x;
double y;

printf("Entrer un entier\n");
scanf("%d", &i); /* passer l'adresse */
printf("La valeur de i est %d\n", i);
printf("Entrer un réel : float, double \n");
scanf("%g %lg", &x, &y); /* passer les adresses */
printf("Les valeurs de x et y sont %g et %g\n",
x, y);

exit(EXIT_SUCCESS);
}
```

```
program read_write

implicit none
integer :: i
real :: x

write(*,*) "Entrer un entier"
read(*,*) i
write(*,*) "La valeur de i est ", i
write(*,*) "Entrer un réel "
read(*,*) x
write(*,*) "La valeur de x est ", x

end program read_write
```


4.1.1 Introduction aux formats en C

Attention : quelques différences entre **scanf** (type exact) et **printf** (**conversion** de type possible car passage d'argument par copie)

En sortie avec **printf**

Type	Format
char	%c
chaîne	%s
short (converti en)/ int	%d
unsigned int/unsigned short	%ud (%o, %x)
long	%ld
unsigned long	%lu (%lo, %lx)
long long	%lld
unsigned long long	%llu
float (convertis en)/double	(%e, %f) %g
long double	(%Le, %Lf) %Lg



En entrée avec `scanf`

Type	Format
char	<code>%c</code>
short	<code>%hd</code>
unsigned short	<code>%hu</code> (<code>%ho</code> , <code>%hx</code>)
int	<code>%d</code>
unsigned int	<code>%u</code> (<code>%o</code> , <code>%x</code>)
long	<code>%ld</code>
unsigned long	<code>%lu</code> (<code>%lo</code> , <code>%lx</code>)
long long	<code>%lld</code>
unsigned long long	<code>%llu</code> (<code>%llo</code> , <code>%llx</code>)
float	<code>(%e, %f) %g</code>
double	<code>(%le, %lf) %lg</code>
long double	<code>(%Le, %Lf) %Lg</code>



5 Structures de contrôle

Par défaut, exécution séquentielle des instructions une seule fois, dans l'ordre spécifié par le programme.

⇒ trop restrictif

Introduire des **structures de contrôles** (*flow control*) permettant de modifier le cheminement lors de l'exécution des instructions :

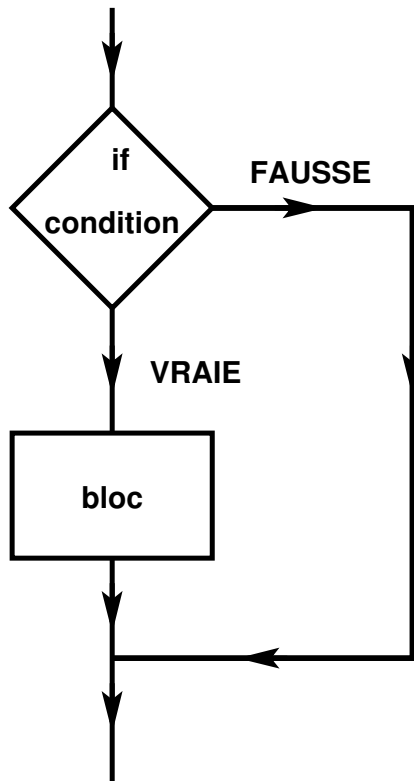
- exécution **conditionnelle** (**if**) ou **aiguillage** (**case**) dans les instructions
- **itération** de certains blocs (**for**, **do**, **while**...)
- **branchements** (**cycle** ou **continue**, **exit** ou **break**, ...)

⇒ Mettre en évidence les blocs par **indentation du code** (cf python)

Nommage possible des structures en fortran (utile pour les branchements)

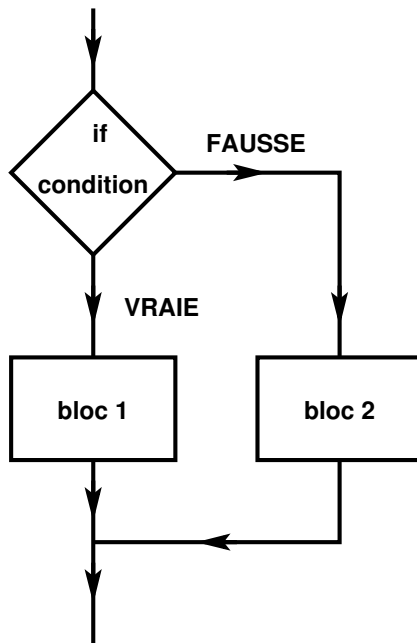
5.1 Structure conditionnelle `if`

5.1.1 Condition `if`



c	fortran 90
if (expression) instruction	if (expr. log.) instruction
if (expression) { bloc d'instructions }	if (expr. log.) then bloc d'instructions end if
expression testée	
entier vrai si non null en C89	de type booléen

5.1.2 Alternative `if ... else`



c	fortran 90
<pre> if (expression) { bloc d'instructions 1 } else { bloc d'instructions 2 } </pre>	<pre> if (expr. log.) then bloc d'instructions 1 else bloc d'instructions 2 end if </pre>

5.1.3 Exemples d'alternative if ... else

```
#include <stdio.h> /* fichier if2.c */
#include <stdlib.h>
int main(void)
{ /* structure if ... else */
  int i, j, max ;
  printf("entrer i et j (entiers)\n") ;
  scanf("%d %d", &i, &j) ;
  if (i >= j) { /* affichage du max de 2 nombres */
    printf(" i >= j \n") ;
    max = i ; /* bloc d'instructions */
  }
  else { /* préférer ajouter { */
    max = j ; /* instruction simple */
  } /* préférer ajouter } */
  printf(" i= %d, j= %d, max = %d\n", i, j, max);
  exit(EXIT_SUCCESS) ;
}
```

```
! structure if then ... else ... endif  
! affichage du max de deux nombres
```

```
program alternative
```

```
implicit none
```

```
integer :: i, j, maxij
```

```
write(* , *) "entrer i et j (entiers)"
```

```
read(* , *) i, j
```

```
if (i >= j) then
```

```
  write(* , *) "i >= j "
```

```
  maxij = i
```

```
! bloc d'instructions
```

```
else
```

```
  maxij = j
```

```
! instruction simple
```

```
end if
```

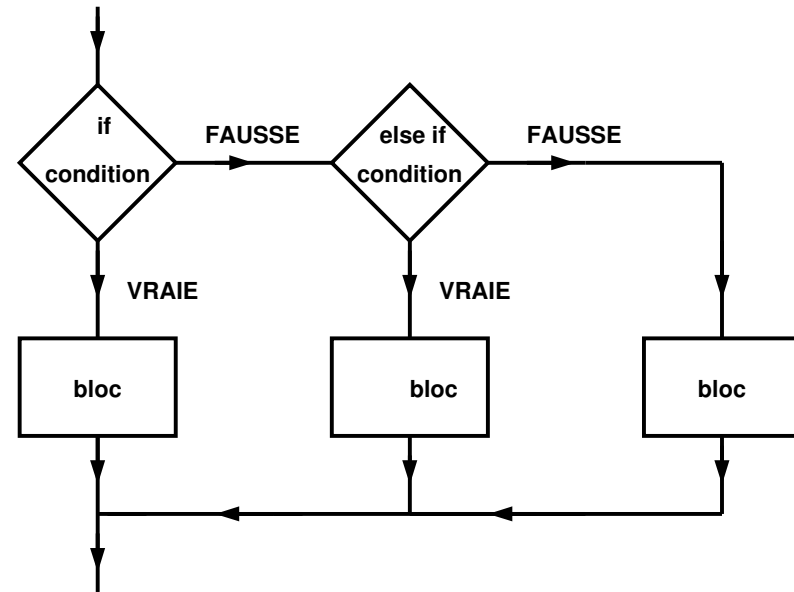
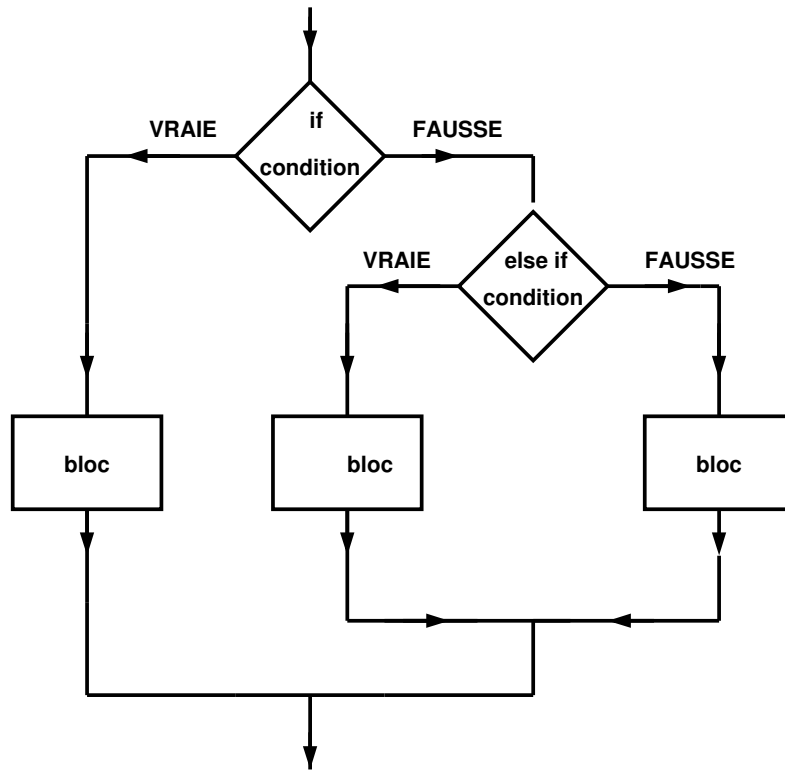
```
write(* , *) "i =", i, ", j =", j, ", max =", maxij
```

```
end program alternative
```

5.1.4 Alternatives imbriquées `if ... else`

Imbrication simple \Rightarrow deux **end if**

Fusion des retours \Rightarrow un **end if**



5.1.5 Cas de else if en fortran

```

program imbrication
implicit none
integer :: i
write(*,*) "entrer i entier"
read(*,*) i ! deux if imbriqués
if (i < -10) then ! if externe
    write(*,*) "i < -10"
else
    if (i < 10) then ! if interne
        write(*,*) "-10 <= i < 10"
    else
        write(*,*) "i >= 10 "
    end if ! end if interne
end if ! end if externe
end program imbrication


```

```

! structure avec else if
program imbrication
implicit none
integer :: i
write(*,*) "entrer i entier"
read(*,*) i
if (i < -10) then
    write(*,*) "i < -10"
! else if sur une même ligne
else if (i < 10) then
    write(*,*) "-10 <= i < 10"
else
    write(*,*) "i >= 10 "
end if ! un seul end if
end program imbrication

```

5.2 Aiguillage avec switch/case (pas avec des flottants)

C	fortran 90
<pre> switch (expr. entière) { case sélecteur1 : bloc d'instructions [break ;] case sélecteur2 : bloc d'instructions [break ;] ... default : bloc d'instructions } </pre>	<pre> select case (expr.) case (sélecteur1) bloc d'instructions case (sélecteur2) bloc d'instructions ... case default bloc d'instructions end select </pre>
sélecteur	
expression constante entière ou caractère	expression constante entière ou caractère ou liste de constantes séparées par virgules, ou intervalle fini ou semi-infini, (bornes séparées par le caractère :)
 Sans break , on teste tous les cas qui suivent le premier sélecteur vrai !	

5.2.1 Exemples d'aiguillage case

```
#include <stdio.h>                                /* fichier case.c */
#include <stdlib.h>

/* structure switch case sur des entiers          */
int main(void)
{
    int i ;
    printf(" entrer un entier : ") ;
    scanf("%d", &i) ;
    printf("\n i = %d \n", i) ;
    switch (i)
    {
        case 0 :
            printf(" i vaut 0 \n") ;
    }
    /* début de bloc */
}
```

```
    break;                /* nécessaire ici ! */
case 1 :
    printf(" i vaut 1 \n") ;
    break;                /* nécessaire ici ! */
default :
    printf(" i différent de 0 et de 1 \n") ;
}                          /* fin de bloc */
exit(EXIT_SUCCESS) ;
}
```

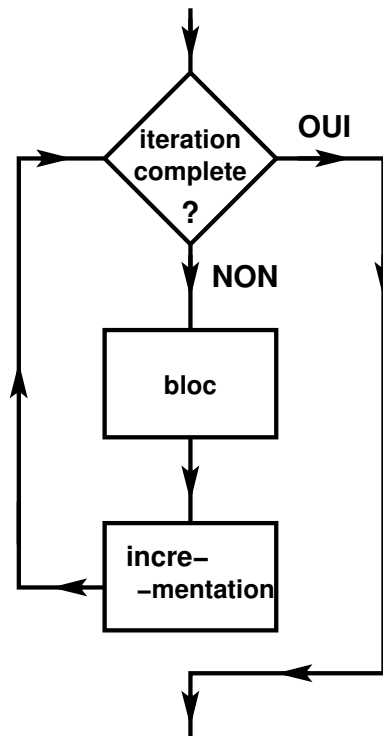
```
! structure select case sur des entiers fichier case.f90
PROGRAM choix
IMPLICIT NONE
INTEGER :: i
WRITE (*,*) " entrer un entier : "
READ (*,*) i ! lecture
WRITE (*,*) " i = ", i ! vérification
SELECT CASE (i) ! début de bloc
  CASE (0)
    WRITE (*,*) " i vaut 0 "
  CASE (1)
    WRITE (*,*) " i vaut 1 "
  CASE default
    WRITE (*,*) " i différent de 0 et de 1 "
END SELECT ! fin de bloc
END PROGRAM choix
```

```
#include <stdio.h>                                     /* fichier case1.c */
#include <stdlib.h>
/* exemple d'utilisation de la structure case sans break
 * pour "factoriser des cas" et les traiter en commun
 */
int main(void)
{
char c ;
printf("entrer un caractère : est-ce une ponctuation double ?");
scanf("%c", &c) ;
printf("\n caractère = %c \n", c) ;
switch (c)
{
case '?' :
case '!' :
case ';' :
```

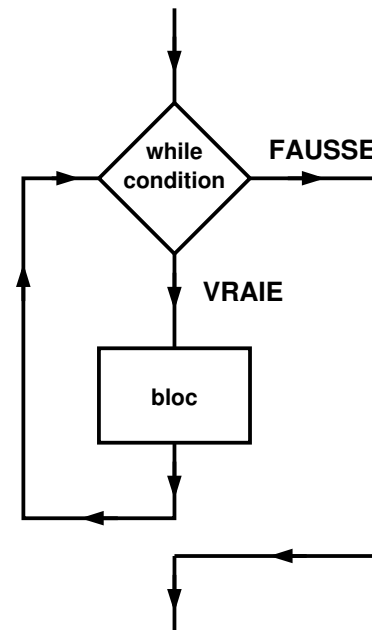
```
case ':' :  
    printf(" ponctuation double \n") ;  
    break ;          /* à la fin des 4 cas */  
default :  
    printf(" autre caractère \n") ;  
}  
exit(EXIT_SUCCESS) ;  
}
```

```
PROGRAM case1 ! fichier case1.f90  
! exemple d'utilisation de la structure select case  
! avec des listes de constantes pour les cas à traiter en commun  
IMPLICIT NONE  
CHARACTER(len=1) :: c  
WRITE(* , *) "entrer un caractère: est-ce une ponctuation double ?"  
READ(* , *) c  
WRITE(* , *) " caractère = ", c  
SELECT CASE (c)  
  CASE ('?' , '!' , ';' , ':')  
    WRITE(* , *) "ponctuation double"  
  CASE default  
    WRITE(* , *) "autre caractère "  
END SELECT  
END PROGRAM case1
```


5.3 Structures itératives ou boucles



nombre d'itérations
connu a priori
structure **for** ou
do avec compteur



nombre d'itérations
inconnu a priori
structure **while**

C		fortran 90
for (expr ₁ ; expr ₂ ; expr ₃) { bloc d'instructions }	boucle (avec compteur en fortran)	do entier = début, fin [, pas] bloc d'instructions end do
while (expr.) { instruction }	tant que faire	do while (expr. log.) bloc d'instructions end do
do { instruction } while (expr.);	faire ... faire ... tant que	

Boucle **for**

- **expr1** exécuté **une fois** avant l'entrée dans la boucle
 (généralement initialisation d'un compteur)
- **expr2** est une condition évaluée à chaque début de répétition
- **expr3** est effectué à la fin de chaque itération
 (généralement incrémentation du compteur)

5.3.1 Exemples de boucle for ou do

```
#include <stdio.h>                /* fichier for.c */
#include <stdlib.h>
/* affichage des entiers impairs < à un entier donné */
/* mise en oeuvre de la structure "for" (5 versions) */
int main(void)
{
    int i, m = 11 ;
    printf("affich. entiers impairs <= %d (5 vers avec for)\n", m) ;

    for (i = 1; i <= m; i = i + 2)
    {
        printf(" %d \n", i) ;      /* un bloc : conseillé */
    }
    printf("-----\n");         /* en dehors du for ! */
}
```

```
for (i = 1; i <= m; i = i + 2)    /* éviter ce qui suit */
    printf(" %d \n", i) ;        /* instruction simple */
printf("-----\n");           /* en dehors du for !! */
for (i = 1; i <= m; printf(" %d \n", i), i=i+2) ;
printf("-----\n");           /* avec une instruction vide ! */
for (i = 1; i <= m; printf(" %d \n", i), i++, i++) ;
    /* avec une autre instruction vide !! noter le ; */
printf("-----\n");
for (i = 1; i <= m; )           /* sans incrémentation ici */
    {                           /* un bloc d'instructions */
    printf(" %d \n", i);
    i++ ;
    i++ ;
    }                           /* fin du bloc */
exit(0) ;
}
```

! affichage des entiers impairs inférieurs à un entier donné
! mise en oeuvre de la structure "do" avec compteur

```
PROGRAM iter                                ! début de programme principal
IMPLICIT NONE                              ! rend les déclarations obligatoires
INTEGER :: i, m = 11                       ! déclarations + initialisation
WRITE (*,*) "affichage des entiers impairs <= ", m

DO i = 1, m, 2                              ! i de 1 à m par pas de 2
    ! début de bloc
    WRITE (*,*) i                          ! bloc réduit à une instruction
    ! fin de bloc
END DO

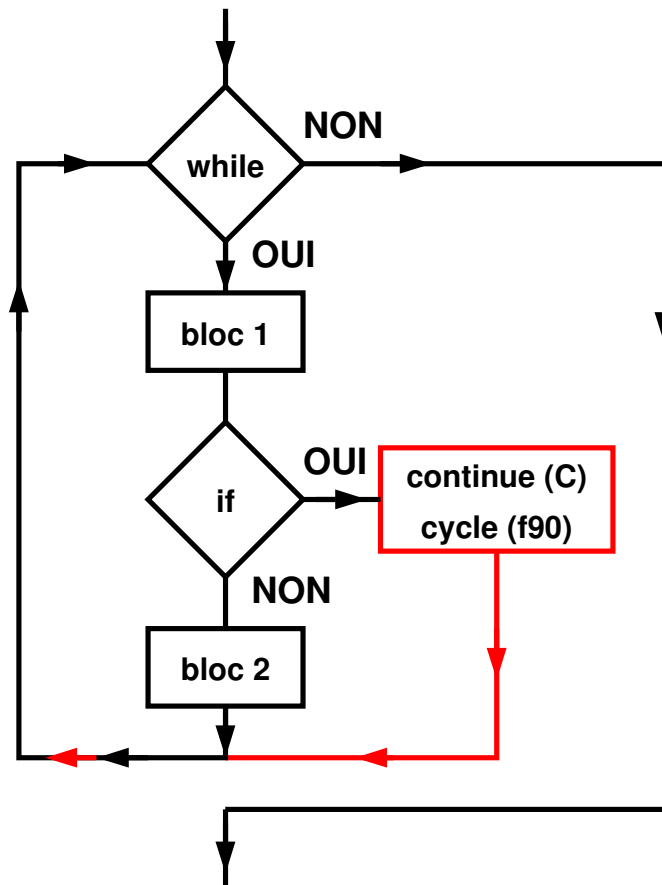
END PROGRAM iter                            ! fin de programme principal
```

5.4 Branchements ou sauts

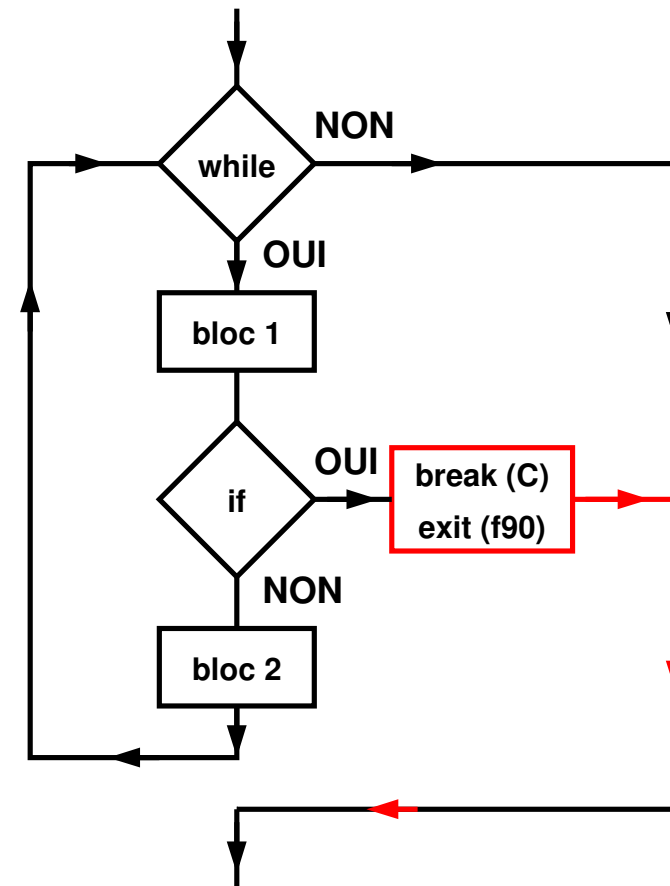
C		fortran 90
continue ;	bouclage anticipé	cycle
break ;	sortie anticipée	exit
goto étiquette ; ^a	branchement (à éviter)	go to étiquette-numérique

a. l'étiquette est un identificateur suivi de **:** en tête d'instruction.

Rebouclage anticipé
continue ou **cycle**



Sortie anticipée de boucle
break ou **exit**



5.4.1 Exemples de bouclage anticipé `cycle/continue`

```
#include <stdio.h>
#include <stdlib.h>
/* recyclage anticipé via "continue" */
int main(void)
{
    int i = 0 , m = 11;
    while ( i < m )
    {
        i++ ;
        if ( (i % 2) == 0 ) continue ; /* rebouclage si i pair */
        printf(" %d \n", i) ;
    }
    exit(0) ;
}
```


! recyclage anticipé via "cycle"

```
PROGRAM recycle
IMPLICIT NONE
INTEGER :: i, m = 11
!  
i = 0
DO WHILE ( i < m )
    i = i + 1           ! modification de la condition du while
    IF ( MOD(i, 2) == 0 ) CYCLE           ! rebouclage si i pair
    WRITE(*,*) i
END DO
!  
END PROGRAM recycle
```

5.4.2 Exemple de sortie de boucle `break/exit`

```
#include <stdio.h>
#include <stdlib.h>
/* sortie anticipée de boucle via "break" */
int main(void)
{
int i = -1 , m = 11;
while ( i < 100 )
    {
    i += 2 ;
    if ( i > m ) break ; /* sortie de boucle */
    printf(" %d \n", i) ;
    }
exit(0) ;
}
```

```
! sortie anticipée via "exit"  
! impression des entiers impairs entre 1 et 11  
PROGRAM EXIT  
IMPLICIT NONE  
INTEGER :: i = -1 , m = 11  
!  
DO ! boucle infinie a priori  
    i = i + 2  
    IF ( i > m ) EXIT ! sortie anticipée dès que i >= m  
    WRITE (*,*) i  
END DO  
!  
END PROGRAM EXIT
```

6 Introduction aux pointeurs

6.1 Intérêt des pointeurs

- **variables** permettant de désigner successivement différentes variables afin de :
 - ⇒ faire intervenir un niveau supplémentaire de paramétrage dans la manipulation des données : action **indirecte** sur une variable
 - ⇒ créer ou supprimer des variables lors de l'exécution : **allocation dynamique**
- différences notables entre pointeurs en C et en fortran

indispensables en C	absents en fortran 77,
pour les arguments des fonctions	mais introduits en fortran 90/95
et les tableaux dynamiques	et étendus en fortran 2003
stockent des adresses des variables	pas d'accès aux adresses
- utilisation commune : tris sur des données volumineuses, implémentation de structures de données autoréférencées (listes chaînées par ex.)

6.2 Pointeurs et variables : exemple du C

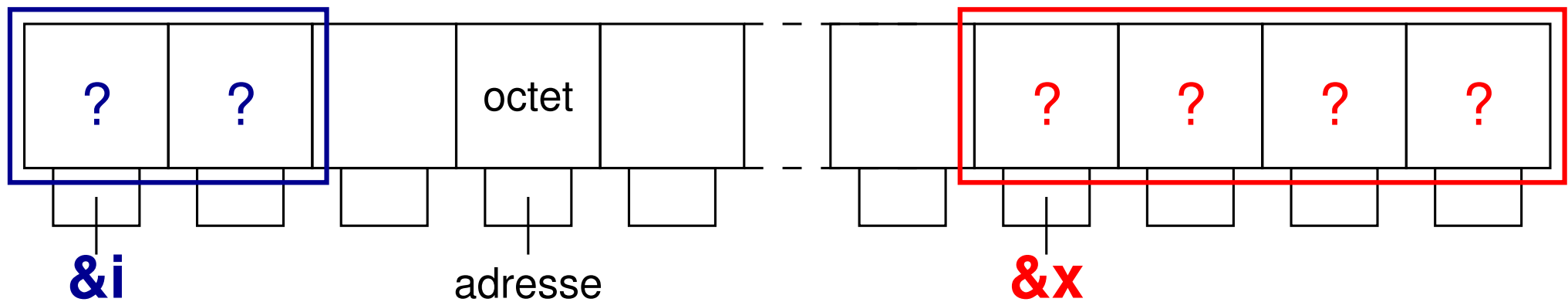
Déclarer une variable d'un **type** donné =
réserver une zone mémoire dont
la **taille** (`sizeof` en C) dépend du type
et le **codage** est fixé par le type

sizeof (short int)

sizeof (float)

short int i ;

float x ;



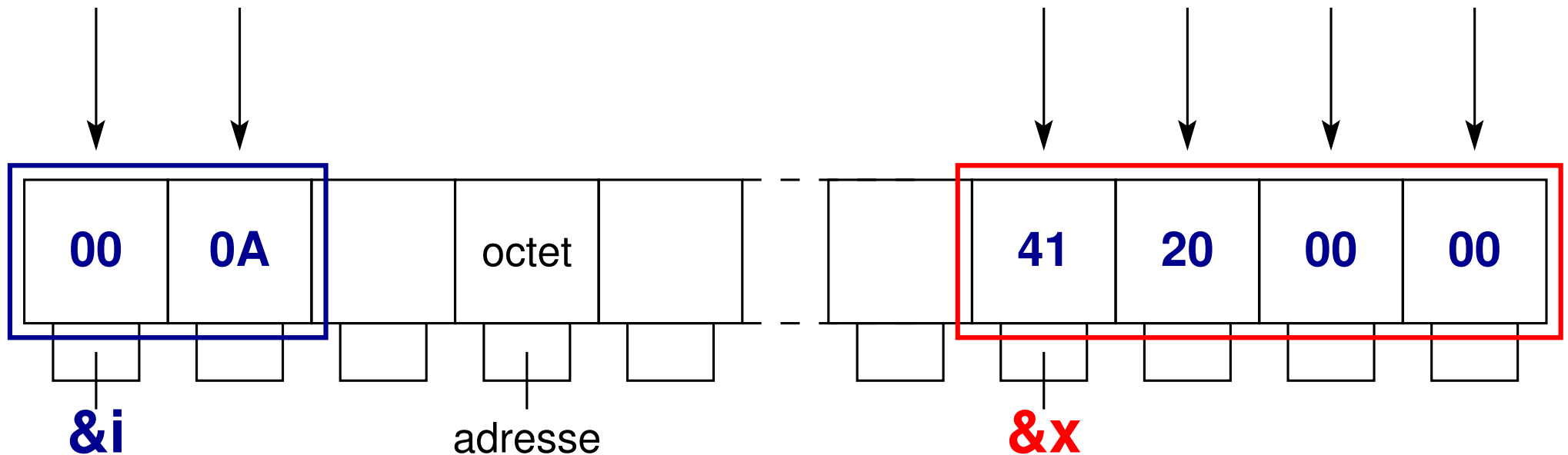
**Affecter une valeur à une variable d'un type donné =
écrire la valeur dans les cases réservées selon le codage du type**

sizeof (short int)

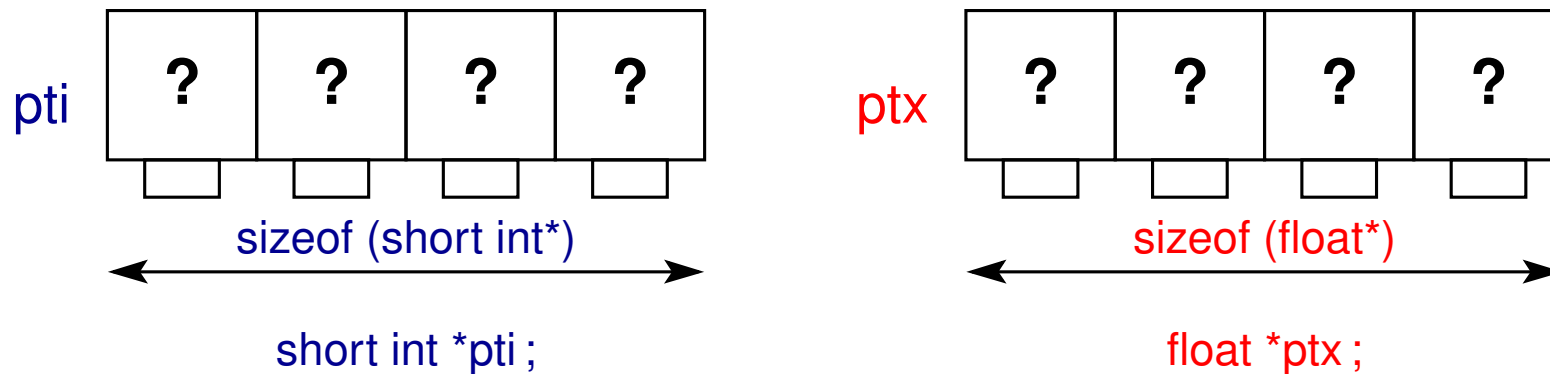
sizeof (float)

i=10;

x=10.f;



Déclarer une variable pointeur vers un **type donné =**
réserver une zone mémoire pour stocker **des adresses**
de variables de ce type : **les pointeurs sont typés**
⇒ leur **type** indique la **taille** et le **codage** de la **cible** potentielle



La taille du pointeur est indépendante du type pointé
Elle dépend du processeur (32/64 bits).

6.2.1 Affectation d'un pointeur en C

Affecter l'adresse d'une variable à un pointeur : `pti = &i; ptx = &x;`

= copier l'adresse mémoire de la variable cible (opérateur `&`)

dans la zone mémoire réservée lors de la déclaration du pointeur.

le pointeur `pti` **pointe sur** la variable `i`,
la variable `i` **est la cible du** pointeur `pti`.

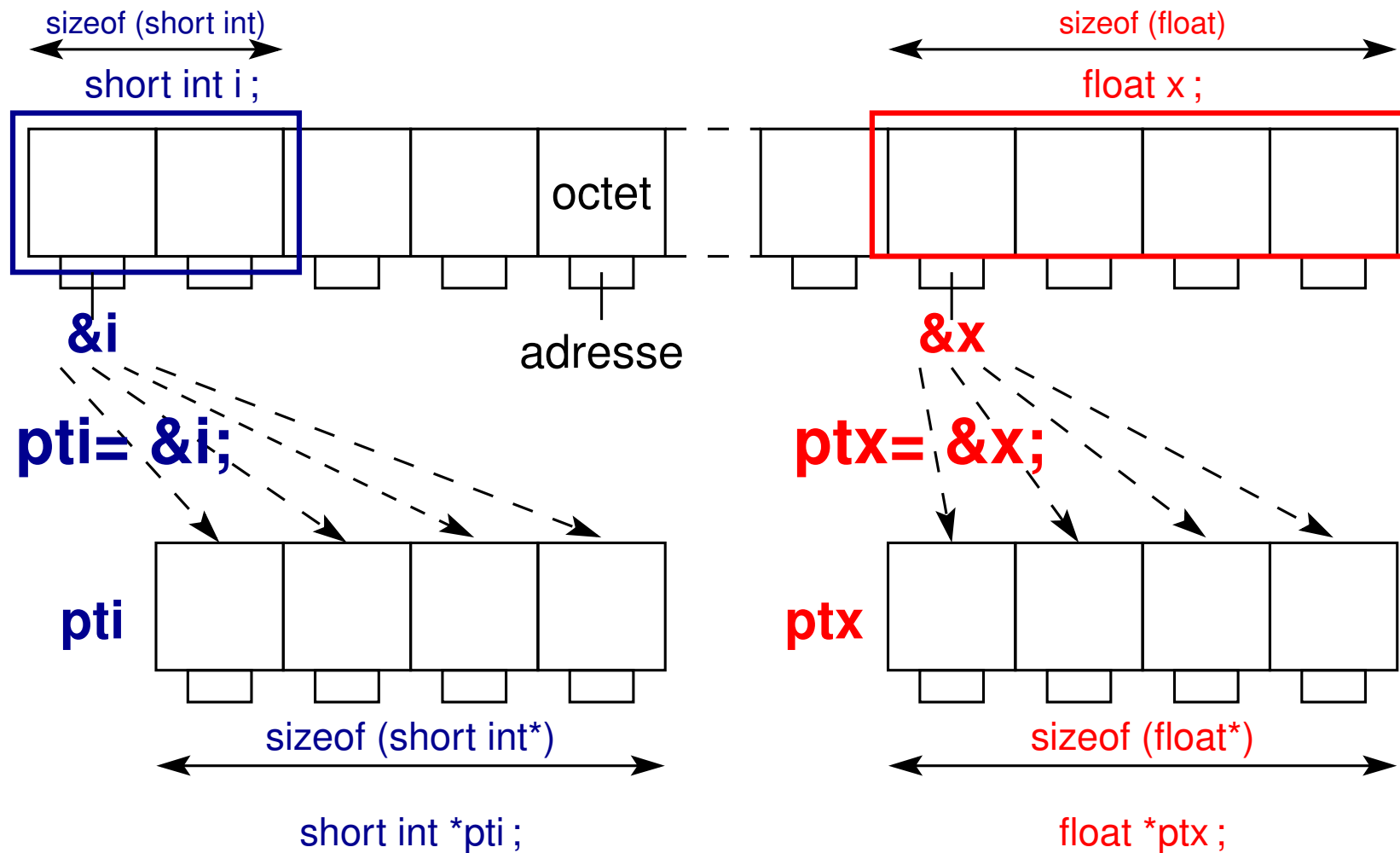
Attention :

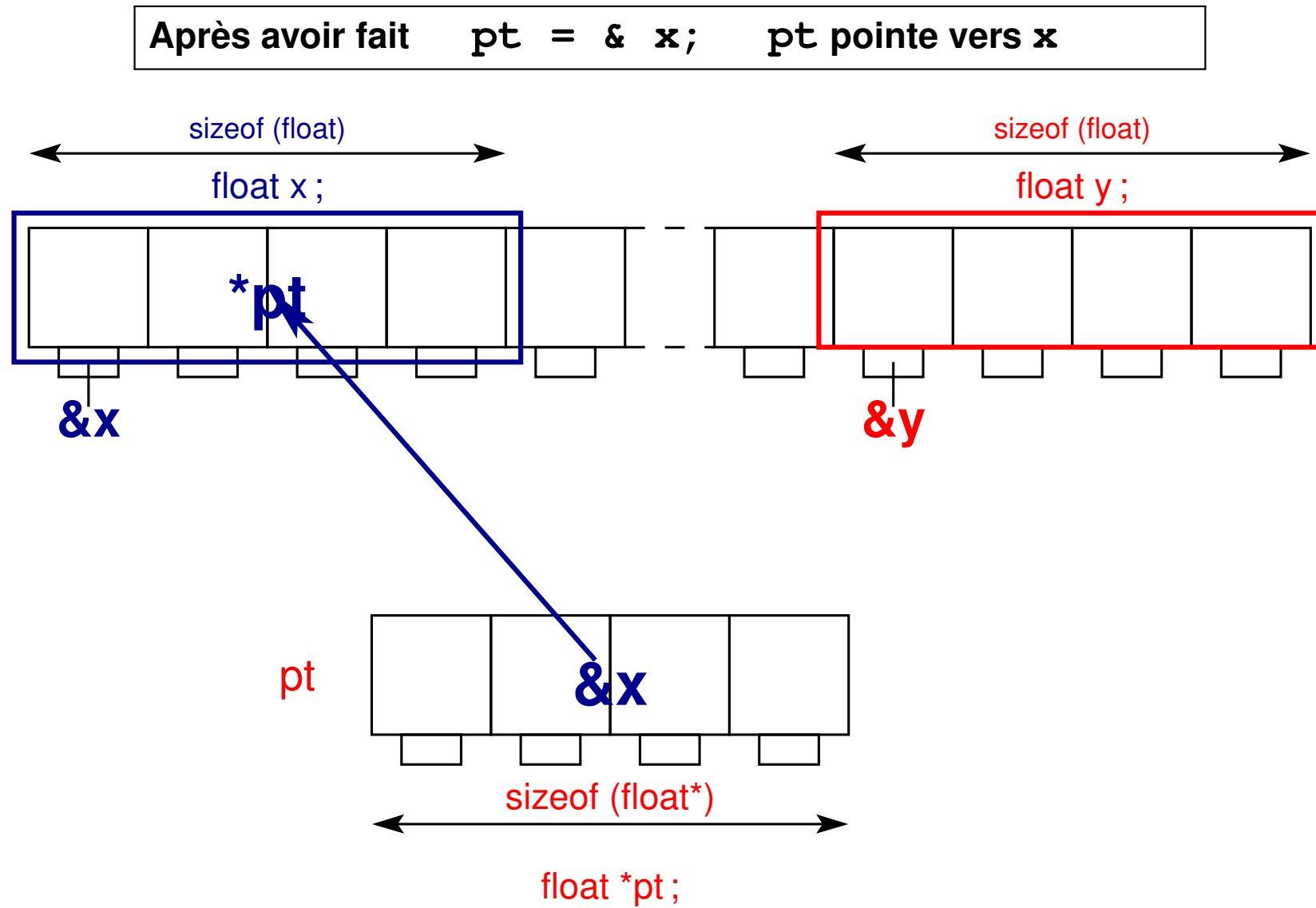
- il faut que les variables **cibles** `i` et `x` aient été **déclarées au préalable**.
 - comme pour une variable ordinaire, l'adresse contenue dans le pointeur est indéterminée avant l'affectation du pointeur
- ⇒ **initialiser un pointeur avant de le manipuler**

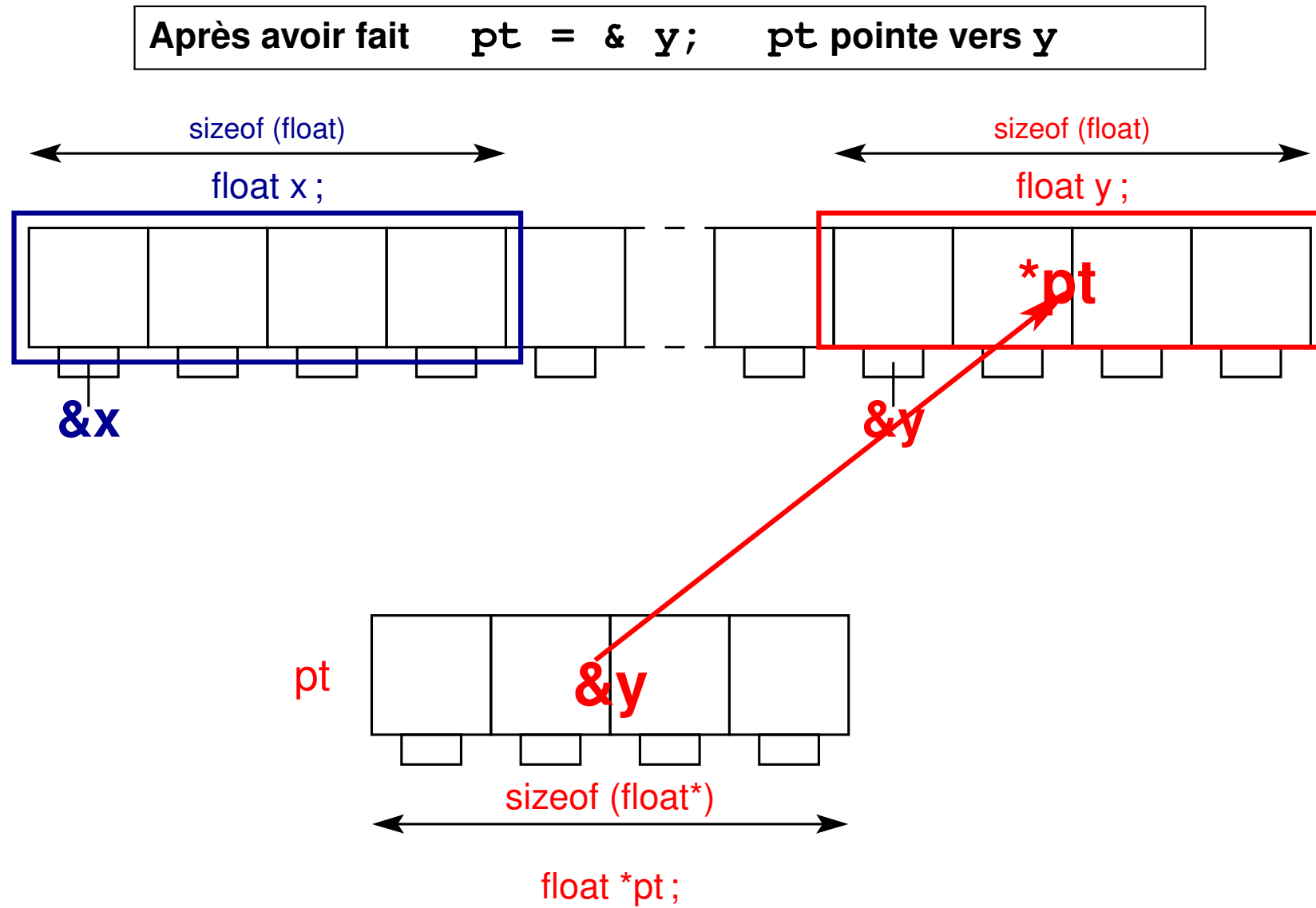
Affecter la **valeur d'un pointeur** `pty` à un pointeur de **même type** `ptx` :

`ptx = pty ;` (recopie d'adresse)

**Faire pointer un pointeur vers une variable =
écrire l'adresse de la variable dans les cases réservées pour le pointeur
= affecter une valeur à la variable pointeur**







6.2.2 Indirection (opérateur * en C)

L'indirection via l'opérateur `*` permet d'**accéder au contenu de la variable pointée** via le pointeur (donc indirectement).

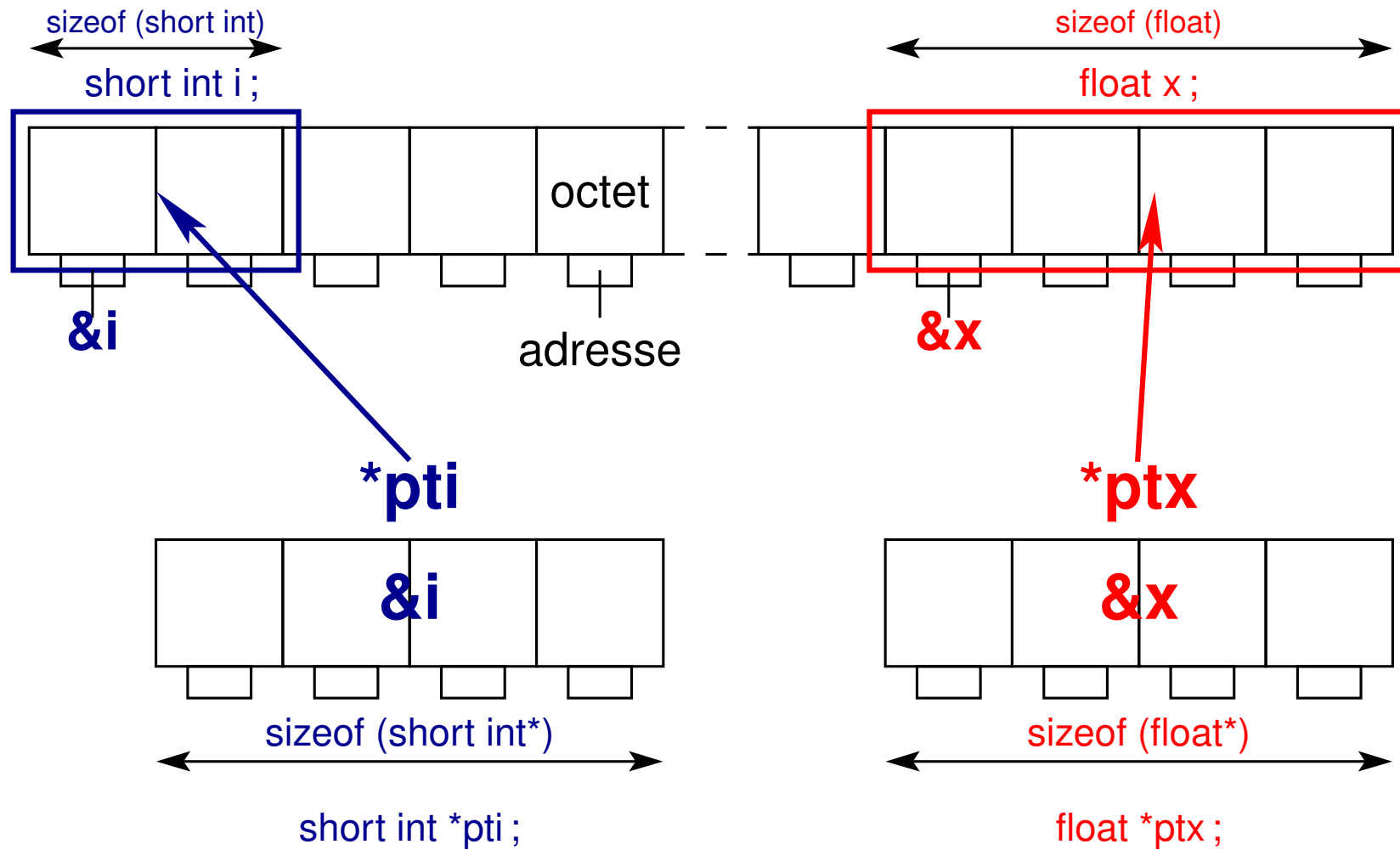
`j = *pti;` la variable `j` prend la valeur de la cible de `pti`

`*pti = 4;` la cible de `pti` vaut désormais 4

Attention :

- pour que le codage/décodage soit correct, il faut que le pointeur soit un pointeur vers une variable de **même type que la cible**.
- affectation de la cible désastreuse si le pointeur n'a pas été initialisé
 - ⇒ modification possible d'une autre variable (erreur aléatoire sournoise)
 - ou accès à une zone mémoire interdite (**segmentation fault** : mieux !)


Opération d'**indirection** (*) sur un pointeur =
accès au contenu des cases pointées (= de la cible)



6.3 Pointeurs en fortran

- Notion **plus haut niveau** qu'en C : pas d'accès aux adresses !
- Pointeur considéré comme un **alias de la cible** : le pointeur désigne la cible
⇒ pas d'opérateur d'indirection
- Pas de type pointeur : **pointer** est un **simple attribut** d'une variable
- **Association** d'un pointeur à une cible par l'opérateur **=>**
Exemple : **ptx => x** signifie `ptx` pointe vers `x`
- **Désassociation** d'un pointeur **ptx => null()** ou **nullify(ptx)**
- Fonction booléenne d'interrogation **associated**
- Mais les **cibles potentielles** doivent posséder :
 - l'attribut **target** (cible ultime)
 - ou l'attribut **pointer** (cas des listes chaînées)

6.4 Syntaxe des pointeurs (C et fortran)

Langage C		Fortran 90
type *ptr;	déclaration	type, pointer :: ptr
type *ptr=NULL;	avec initialisation	type, pointer :: ptr=>null()
type var; (pas d'attribut)	cible	type, target (nécessaire) :: var
ptr = &var ;	pointer sur	ptr => var (associer ptr à var)
*ptr	variable pointée par	ptr
ptr = NULL ;	dissocier	nullify(ptr); ptr=>null()
	associé ?	associated(ptr)
	à la cible var ?	associated(ptr, var)
 concerne les adresses !	ptr2 = ptr1	concerne les cibles !

6.5 Exemples élémentaires (C et fortran)

```
/* usage élémentaire d'un pointeur */
#include <stdio.h>
#include <stdlib.h>
int main(void) {
int i=1, j=2;
int *pi = NULL; // initialisé
pi = &i; // pi devient l'adresse de l'entier i
printf("%d\n", *pi) ; // affiche i
pi = &j; // pi devient l'adresse de l'entier j
printf("%d\n", *pi) ; // affiche j
*pi= 5; // affectation de la cible de pi, soit j
printf("i=%d, j=%d, *pi=%d\n", i, j, *pi) ;
exit(EXIT_SUCCESS);
}
```



```
PROGRAM pointeur ! usage élémentaire d'un pointeur
IMPLICIT NONE
INTEGER, TARGET :: i, j ! target obligatoire pour pointer vers
INTEGER, POINTER :: pi => NULL()
i = 1
j = 2
! association entre pointeur et cible
pi => i ! pi pointe sur i
! affectation
WRITE(*,*) "i=", i, " j=", j, " pi= ", pi
pi => j ! maintenant pi pointe sur j et plus sur i
WRITE(*,*) "i=", i, " j=", j, " pi= ", pi
pi = -5 ! modif de j via le pointeur
WRITE(*,*) "i=", i, " j=", j, " pi= ", pi
END PROGRAM pointeur
```

6.6 Tailles des types de base et adresses en C

Processeur 32 bits AMD

Tailles en octets via sizeof :

char = 1 short int = 2 int = 4 long int = 4

Adresses converties en unsigned long int :

c =3219368569 s =3219368562 i =3219368548 l =3219368536
 &c[0]=3219368569 &s[0]=3219368562 &i[0]=3219368548 &l[0]=3219368536
 &c[1]=3219368570 &s[1]=3219368564 &i[1]=3219368552 &l[1]=3219368540
 &c[2]=3219368571 &s[2]=3219368566 &i[2]=3219368556 &l[2]=3219368544

Tailles en octets via sizeof :

float = 4 double = 8 long double = 12 int * = 4

Adresses converties en unsigned long int :

f =3219368524 d =3219368496 L =3219368448 pi =3219368436
 &f[0]=3219368524 &d[0]=3219368496 &L[0]=3219368448 &pi[0]=3219368436
 &f[1]=3219368528 &d[1]=3219368504 &L[1]=3219368460 &pi[1]=3219368440
 &f[2]=3219368532 &d[2]=3219368512 &L[2]=3219368472 &pi[2]=3219368444

Processeur 64 bits AMD

Tailles en octets via sizeof :

char = 1 short int = 2 int = 4 long int = 8

Adresses converties en unsigned long int :

c =140735477551008 s =140735477550992 i =140735477550976 l =140735477550944
 &c[0]=140735477551008 &s[0]=140735477550992 &i[0]=140735477550976 &l[0]=140735477550944
 &c[1]=140735477551009 &s[1]=140735477550994 &i[1]=140735477550980 &l[1]=140735477550952
 &c[2]=140735477551010 &s[2]=140735477550996 &i[2]=140735477550984 &l[2]=140735477550960

Tailles en octets via sizeof :

float = 4 double = 8 long double = 16 int * = 8

Adresses converties en unsigned long int :

f =140735477550928 d =140735477550896 L =140735477550848 pi =140735477550816
 &f[0]=140735477550928 &d[0]=140735477550896 &L[0]=140735477550848 &pi[0]=140735477550816
 &f[1]=140735477550932 &d[1]=140735477550904 &L[1]=140735477550864 &pi[1]=140735477550824
 &f[2]=140735477550936 &d[2]=140735477550912 &L[2]=140735477550880 &pi[2]=140735477550832

6.7 Initialiser les pointeurs !

Déclarer un pointeur ne réserve pas de mémoire pour la zone pointée !

L'adresse qu'il contient est aléatoire

⇒ le résultat d'une indirection est aussi aléatoire

⇒ initialiser les pointeurs à **NULL** ou **null ()**

pour forcer une erreur en cas d'indirection sur un pointeur non associé.

```

/* pointeur non initialisé => erreur aléatoire d'accès mémoire */
#include <stdio.h>
#include <stdlib.h>
int main(void) { // fichier erreur-pointeur.c
int i;
int *pi, *pj;
pi = &i; /* pi devient l'adresse de l'entier i */
/* affichage des valeurs des pointeurs pi et pj */
printf("valeurs des pointeurs = adresses\n");
printf("pi=%019lu\npj=%019lu\n", (unsigned long int) pi,
                                           (unsigned long int) pj);

i = 1;
/* suivant l'ordre de decl. *pi, *pj => ? mais *pj, *pi erreur */
printf("i=%d, *pi=%d , *pj=%d\n", i, *pi, *pj); /* aleatoire*/
/* si l'affichage se fait, il est aleatoire (statique/dynamique) */
=> *pj = 2; /* => erreur fatale en écriture */
/* accès à une zone interdite quand on affecte 2 à l'adresse pj */
exit(EXIT_SUCCESS);
}

```

```

PROGRAM erreur_pointeur
IMPLICIT NONE
INTEGER, TARGET :: i, j ! target obligatoire pour pointer vers i et j
INTEGER, POINTER :: pi => null() ! spécifie absence d'association de pi
! sinon état indéterminé du pointeur par défaut -> le préciser
j = 2
WRITE(*,*) "au départ : pi associé ?", associated(pi)
! l'instruction suivante provoque un accès mémoire interdit
=> pi = 3 ! affectation d'une valeur à un pointeur non associé : erreur
! il faut d'abord associer pi à une variable cible comme suit
pi => i ! associe pi à i dont la valeur est pour le moment indéterminée
WRITE(*,*) "après pi=>i : pi associé ?", associated(pi)
WRITE(*,*) "après pi=>i : pi associé à i ?", associated(pi,i)
i = 1 ! donc pi vaudra 1
WRITE(*,*) "après i=1 : i= ", i, " j= ", j, " pi = ", pi
pi = 10 * pi ! on modifie en fait la cible pointée par pi
WRITE(*,*) "après pi=10*pi : i= ", i, " j= ", j, " pi = ", pi
pi => j ! pi pointe maintenant sur j qui vaut 2
WRITE(*,*) "après pi=>j : pi associé à i ?", associated(pi,i)
WRITE(*,*) "après pi=>j : i= ", i, " j= ", j, " pi = ", pi
END PROGRAM erreur_pointeur

```

7 Procédures : fonctions et sous-programmes

7.1 Généralités

Procédures = **fonctions** (C et fortran) ou **sous-programmes** (fortran)

Intérêt :

- **factorisation** d'instructions répétitives \Rightarrow code plus lisible et modulaire
- **paramétrage** de certaines opérations \Rightarrow arguments de la procédure

Distinguer :

- Argument **formel** ou **muet** (*dummy*) dans la définition de la procédure
- Argument **effectif** (*actual*) dans l'appelant

Langage C		Fortran 90
Seulement des fonctions		fonctions et sous-programmes

fonctions typées avec retour (analogue des fonctions mathématiques)		
return expression ; arguments passés par copie	calcul de la valeur de retour dans la fonction	fct = expression (éviter de modifier les arguments)
=fct (x1, x2, ...) a=f1 (x) +2*f1 (x/2) ;	invocation dans l'appelant dans des expressions	=fct (x1, x2, ...) a=f1 (x) +2*f1 (x/2)

procédures à effets de bord (<i>side effects</i>)		
fonctions de type void return ; (sans expression)	entrées/sorties par ex. ou modif des paramètres (fortran)	sous-programmes (<i>subroutines</i>)
fct (x1, x2, ...) ;	invocation dans l'appelant	CALL sub (x1, x2, ...)

7.1.1 Structure d'un programme C

- Un programme C = une ou plusieurs **fonctions** dont au moins la fonction **main** : le programme principal.

N.-B : à l'extérieur de ces fonctions, il peut comporter des déclarations de variables, des déclarations de fonctions spécifiant leur prototype (interface en fortran), et des directives pour le préprocesseur introduites par **#**.

- La définition d'une **fonction** se compose d'un **entête** et d'un **corps** (entre **{** et **}**) qui est en fait une instruction composée.
- L'**entête** d'une fonction spécifie le **type** de la valeur de retour, le nom de la fonction et ses paramètres ou arguments :

type valeur_de_retour (type1 arg1, type2 arg2, ...)

où chaque argument est déclaré par son type, suivi de son identificateur

- Le **corps** de la fonction doit se terminer par **return expression;** pour renvoyer une valeur (ou rien si **void**) à l'appelant
- Pas d'imbrication des fonctions en C : définition en dehors de toute fonction (y compris du `main`)

Déclarer une fonction avant utilisation = indiquer son **prototype** (l'entête suivi de **;**)
permet au compilateur :

- de vérifier la concordance de l'appel en **nombre de arguments**
- d'effectuer les **conversions éventuelles** des arguments effectifs à l'appel vers le type des arguments formels déclarés et au retour dans l'expression appelante

Par ordre de préférence croissante :

- une définition tient lieu de déclaration
mais elle doit être placée **avant l'appel**
- déclarer dans la fonction appelante (rare car portée limitée)
- déclarer en global \Rightarrow visibilité dans tout ce qui suit dans le fichier
 \Rightarrow déclarer en début de fichier
- inclure des **fichiers d'entête** de suffixe **.h** contenant les prototypes
#include "ma_fct.h" au début des fichiers source
de définition de la fonction et des fonctions appelantes

7.1.2 Structure générale d'un programme fortran

Un **programme** fortran = un programme principal

```
program identificateur
```

...

```
end program identificateur
```

plus éventuellement des **procédures** :

- **fonctions** avec valeur de retour typée :

```
[type] function identificateur(arg1, arg2, ...)
```

...

```
end function identificateur
```

appelées quand elles apparaissent dans une expression

- **sous-programmes** :

```
subroutine identificateur(arg1, arg2, ...)
```

...

```
end subroutine identificateur
```

appel explicite par **call** *identificateur*(*arg1*, *arg2*, ...)

Les **procédures** peuvent être :

1. **internes** à un programme ou une autre procédure :

introduites par **contains** et non réutilisables

2. **externes** : (comme en fortran 77)

N.B. : une procédure peut être externe mais dans le même fichier que l'appelant

⇒ compilation séparée sans contrôle interprocédural

⇒ fournir l'**interface** à l'appelant pour permettre ce contrôle

⇒ duplication de code comme en C ... risque d'incohérences

3. **intégrées dans des modules externes** (introduites par **contains**)

⇒ l'interface est mémorisée dans un fichier de module lors de la compilation

⇒ **use nom_module** permet à l'appelant de connaître l'interface en

relisant le fichier de module **nom_module.mod**

⇒ **solution la plus portable**

NB : à l'extérieur des programmes ou procédures, pas de déclarations de variables
sauf variables globales dans un module (avant `contains`)

7.1.3 Fonctions renvoyant une valeur à l'appelant

Définition de la fonction **fct**

Langage C	Fortran 90
<pre> typedef fct (type1 x1, ...) /* déclaration des arg. dans l'entête */ { /* corps de la fonction */ <i>typeloc</i> varloc ; /* variable(s) locale(s) */ ... return <i>expression</i> ; /* résultat de type typedef */ } </pre>	<pre> typedef FUNCTION fct (x1, ...) type1, intent(...) :: x1 ! déclaration des arguments + intent ! corps de la fonction <i>typeloc</i> :: varloc ! variable(s) locale(s) ... fct = <i>expression</i> ! résultat de type <i>typedef</i> RETURN ! optionnel sans argument END FUNCTION fct </pre>

Exemples de fonctions renvoyant une valeur

Dans la **définition** de la fonction **som**, p est une **variable muette (=formelle)**

```

int som(const int p) {
  /*      ^ constant dans la fct */
  /* somme des p premiers entiers */
  int i , s ; /* var. locales */
  s = 0;
  for (i = 0; i <= p; i++){
    s += i ;
  }
  return s ; /* valeur rendue */
}

```

```

INTEGER FUNCTION som(p) ! som_int.f
  INTEGER, INTENT(in) :: p
  ! somme des p premiers entiers
  INTEGER :: i, s ! var. locales
  s = 0
  DO i = 0, p
    s = s + i
  END DO
  som = s ! valeur rendue
END FUNCTION som

```

```

#include <stdio.h>
#include <stdlib.h>

/* décl. de la fct somme */
int som(const int p) ;

/* fonction principale */
int main(void) {
    int s ; /* local s */
    /* appel de la fct somme */
    s = som(5) ;
    printf("somme de 1 à 5= %d\n", s);
    /* autre appel */
    printf("s de 1 à 9 = %d\n", som(9));
    exit(EXIT_SUCCESS) ;
}

```

```

PROGRAM ppal
  IMPLICIT NONE
  INTERFACE ! < début d'interface
    FUNCTION som(p)
      ! déclaration de la fonction
      INTEGER :: som ! résultat
      INTEGER , INTENT(in) :: p
    END FUNCTION som
    ! fin déclaration de la fonction
  END INTERFACE ! < fin d'interface
  INTEGER :: s ! local
  WRITE(*,*) "somme de 1 à 5"
  s = som(5) ! appel de la fct
  WRITE(*,*) "sommés = ", s, som(9)
  STOP
END PROGRAM ppal

```

7.1.4 Sous-programmes et fonctions C sans retour

Langage C	Fortran 90
<pre> void fct(<i>type1</i> x1, ...) /* déclaration des arg. dans l'entête */ { /* corps de la fonction */ <i>typeloc</i> varloc ; /* variable(s) locale(s) */ /* effet de bord */ ... return ; } </pre>	<pre> SUBROUTINE sub(<i>x1</i>, <i>x2</i>, ...) <i>type1</i>, intent(...) :: <i>x1</i> <i>type2</i>, intent(...) :: <i>x2</i> ! déclaration des arguments ! corps du ss-pgm <i>typeloc</i> :: varloc ! variable(s) locale(s) ! modif des arg. ou effet de bord ... RETURN END SUBROUTINE sub </pre>

7.1.5 Passage des arguments : nécessité des pointeurs en C

	Langage C		Fortran 90
	déclarer types dans l'entête	arguments	déclarer dans l'interface
⚠	par valeur (copie locale) ⇒ pas de modification possible au retour dans l'appelant ⇒ passer l'adresse par valeur (pointeur) si on veut modifier	passage d'arguments	par référence ⇒ modification possible au retour dans l'appelant mais attributs INTENT spécifiant leur vocation
	par valeur + const passage par pointeur passage par pointeur	entrée sortie modifiable	INTENT (in) INTENT (out) INTENT (inout)
⚠	conversion implicite lors de la copie	entre définition (argument formel) et appel (arg. effectif)	respecter exactement le type (et sous-type, mais \exists procédures génériques)

⇒ différence entre `printf("...", var)` et `scanf("...", &var)`

7.1.6 Vocation des arguments en fortran

Toujours préciser la vocation (**INTENT**) des arguments en fortran

INTENT	IN	OUT	INOUT
argument	d'entrée	de sortie	modifiable
affectation	interdite	nécessaire	possible
argument effectif	expression	variable	variable
contrainte sur	appelé	appelé + appelant	appelant

7.2 Exemples de procédures

7.2.1 Faux échange en C

```
/* Fonctions : passage des arguments par valeur */  
/* => pas de modification en retour */ /* faux-echange.c */  
#include<stdio.h>  
#include<stdlib.h>  
void echange(int a, int b);  
void echange(int a, int b)  
{  
int c; /* variable locale à la fonction */  
printf(" \tdebut echange : %d %d \n", a, b);  
printf(" \tadresses debut echange : %p %p \n", &a, &b);  
c = a;  
a = b;
```

```
b = c;  
printf("\tfin échange : %d %d \n", a, b);  
printf("\tadresses fin échange : %p %p \n", &a, &b);  
return;  
}
```

```
int main(void)  
{  
int n = 1, p = 5;  
printf("avant appel : n=%d p=%d \n", n, p);  
printf("adresses avant appel : %p %p \n", &n, &p);  
échange(n,p); /* appel avec les valeurs */  
printf("après appel : n=%d p=%d \n", n, p);  
printf("adresses après appel : %p %p \n", &n, &p);  
exit(EXIT_SUCCESS) ;  
}
```

En fait, la fonction `echange` travaille sur une **copie locale des variables a et b**.

avant appel : n=1 p=5

adresses avant appel : 0xbf04174 0xbf04170

debut echage : 1 5

adresses debut echage : 0xbf04140 0xbf04144

fin echage : 5 1

adresses fin echage : 0xbf04140 0xbf04144

apres appel : n=1 p=5

adresses après appel : 0xbf04174 0xbf04170

7.2.2 Vrai échange avec pointeurs en C

```
/* Fonctions : passage des adresses en arguments (par valeur,  
/* => modification en retour */    /* fichier echange.c */  
#include<stdio.h>  
#include<stdlib.h>  
void echange(int *pa, int *pb);  
void echange(int *pa, int *pb)  
{ /* pa et pb ^ ^ pointeurs sur des entiers */  
int c; /* variable locale à la fonction (pas pointeur) */  
printf("\tdebut echange : %d %d \n", *pa, *pb);  
printf("\tadresses debut echange : %p %p \n", pa, pb);  
c = *pa;  
*pa = *pb; /* travailler sur les cibles des pointeurs */  
*pb = c;  
printf("\tfin echange : %d %d \n", *pa, *pb);
```

```
printf("\tadresses fin echange : %p %p \n", pa, pb);  
return;  
}
```

```
int main(void)  
{  
int n = 1, p = 5; /* pas de pointeurs ici */  
printf("avant appel : n=%d p=%d \n", n, p);  
printf("adresses avant appel : %p %p \n", &n, &p);  
echange(&n, &p); /* appel avec les adresses */  
printf("apres appel : n=%d p=%d \n", n, p);  
printf("adresses après appel : %p %p \n", &n, &p);  
exit(0) ;  
}
```

Transmettre des copies des **adresses** de **n** et **p** : donc **&n** et **&p**

la fonction `echange` utilise des **pointeurs** pour les recevoir : **pa** et **pb**

elle accède par **indirection** (***pa** et ***pb**) aux **mêmes zones mémoire** que la fonction appelante.

avant appel : n=1 p=5

adresses avant appel : 0xbffbf524 0xbffbf520

debut echange : 1 5

adresses debut echange : 0xbffbf524 0xbffbf520

fin echange : 5 1

adresses fin echange : 0xbffbf524 0xbffbf520

apres appel : n=5 p=1

adresses après appel : 0xbffbf524 0xbffbf520

7.2.3 Procédure de module en fortran 90

```

! Sous-programme avec arguments ! fct3c.f90
! procédure de module + USE => avec contrôle interprocédural
MODULE m_sp
CONTAINS
SUBROUTINE sp(n, p)
  IMPLICIT NONE
  INTEGER, INTENT(in)  :: n ! argument d'entrée non modifiable
  INTEGER, INTENT(out) :: p ! argument de sortie => à affecter
  WRITE(*,*) "      n = ", n, "début de sous-programme"
  p = 2 * n      ! affectation de l'argument de sortie
  WRITE(*,*) "      p = ", p, "fin de sous-programme"
  RETURN        ! rend le contrôle à l'appelant
END SUBROUTINE sp
END MODULE m_sp

```

```
PROGRAM principal
  USE m_sp           ! permet au compilateur d'accéder à l'interface
  IMPLICIT NONE
  INTEGER :: m, q
  m = 5
  WRITE(* , *) "m =", m, "avant appel du sous-programme"
  CALL sp(m, q)      ! appel du sous-programme
  ! use => impossible d'appeler avec un argument réel
  ! call sp(5., q) ! => erreur détectée dès la compilation
  WRITE(* , *) "q =", q, "après appel du sous-programme"
  STOP
END PROGRAM principal
```

7.3 Durée de vie et portée des variables

Durée de vie des variables


variables **permanentes** ou **statiques** : emplacement alloué à la compilation

⇒ durée de vie = celle du programme

variables **temporaires** ou **automatiques** : emplacement alloué sur la pile lors de l'appel de la procédure et (éventuellement) libéré au retour

⇒ pas de mémorisation entre deux appels

Langage C	Fortran
<p style="text-align: center;">variables externes ou globales (\Rightarrow permanentes)</p>	
<p>à l'extérieur de toute fonction, visibles dans les fonctions qui suivent dans le fichier</p>	<p>déclarées dans un module et accessibles via USE (COMMON en fortran 77)</p>
<p style="text-align: center;">variables internes ou locales (\Rightarrow temporaires par défaut) \Rightarrow portée (scope) réduite</p>	
<p>à une fonction ou un bloc en C99, visibles dans la fonction ou le bloc et les blocs inclus Déclarations tardives en C99 portée = de la déclaration à la sortie du bloc</p>	<p>à une procédure, à un module visibles dans la procédure et les sous- procédures internes fortran 2008 : notion de bloc</p>
<p style="text-align: center;">redéclaration locale d'une variable dont la portée assurait la visibilité \Rightarrow masquage par la variable locale</p>	

Langage C	Fortran
Attributs modifiant la durée de vie des variables	
static rend permanente une variable interne à une fonction	save rend permanente une variable locale
 initialisation à chaque appel \nrightarrow permanente (préciser static)	initialisation à la compilation \Rightarrow permanente
Attributs modifiant la portée	
extern rend visible dans un bloc une variable externe déclarée ailleurs (dans un autre fichier)	private/public modifient la portée des variables et procédures de module use mod, only: var1, fct2

7.3.1 Exemples de mauvais usage des variables locales

```
#include<stdio.h>  /* fichier faux-compte.c */
#include<stdlib.h>
/* exemple de communication aléatoire d'information */
/* ni variable globale, ni passage d'argument entre */
/* les fcts avance et recule et la fonction main  */
void avance(void) {
int n;           /* variable locale non initialisée */
n++;
printf("n = %d dans avance\n", n); /* effet de bord */
fprintf(stderr, "\t < adresse de n %p \n", (void *)&n); /* adresse */
return;
}
void recule(void) { /* n partage la mémoire avec p ou q (le 1er déclara
int p;           /* variable locale non initialisée */
int q;           /* variable locale non initialisée */
```

```
p -= 2;
fprintf(stderr, "\t  adresse de p %p >\n", (void *)&p); /* adresse
fprintf(stderr, "\t  adresse de q %p >\n", (void *)&q); /* adresse
return;
}
int main(void) {
    int i;
    for( i = 1; i<=5; i++)
    {
        avance(); /* aucun transfert de paramètre */
        recule(); /* idem mais peut partager la même mémoire */
    }
    exit(0);
}
```

Décompte de 1 si p déclaré avant q

Compte de 1 si q déclaré avant p

Pas de passage explicite de variable

⇒ Comportement aléatoire

7.3.2 Exemples de variable locale permanente avec `static` ou `SAVE`

```
#include<stdio.h>  /* fichier compte-.c */
#include<stdlib.h>
void avance(void) { /* attribut static => permanent */
    static int n=0; /* => une seule initialisation */
    /* int n=0;    => une initialisation à chaque appel ! */
    n++;
    printf("%d \n", n); /* effet de bord */
    return;
}
int main(void) { /* n n'est pas visible dans le main */
    int i;
    for( i = 1; i<=5; i++) {
        avance(); /* aucun transfert de paramètre */
    }
    exit(0);
}
```



```
MODULE m_compte ! fichier compte-.f90
CONTAINS
  SUBROUTINE avance ! sous programme sans argument
    IMPLICIT NONE
    INTEGER, SAVE :: n = 0 ! locale mais statique
    ! INTEGER :: n = 0 ! suffisant car init => statique
    n = n + 1
    WRITE(*,*) n ! effet de bord
  END SUBROUTINE avance
END MODULE m_compte
PROGRAM t_compte
  USE m_compte
  IMPLICIT NONE
  INTEGER :: i
  DO i = 1, 5 ! n est inconnue dans le programme principal
    CALL avance ! appel du sous programme sans argument
  END DO
END PROGRAM t_compte
```

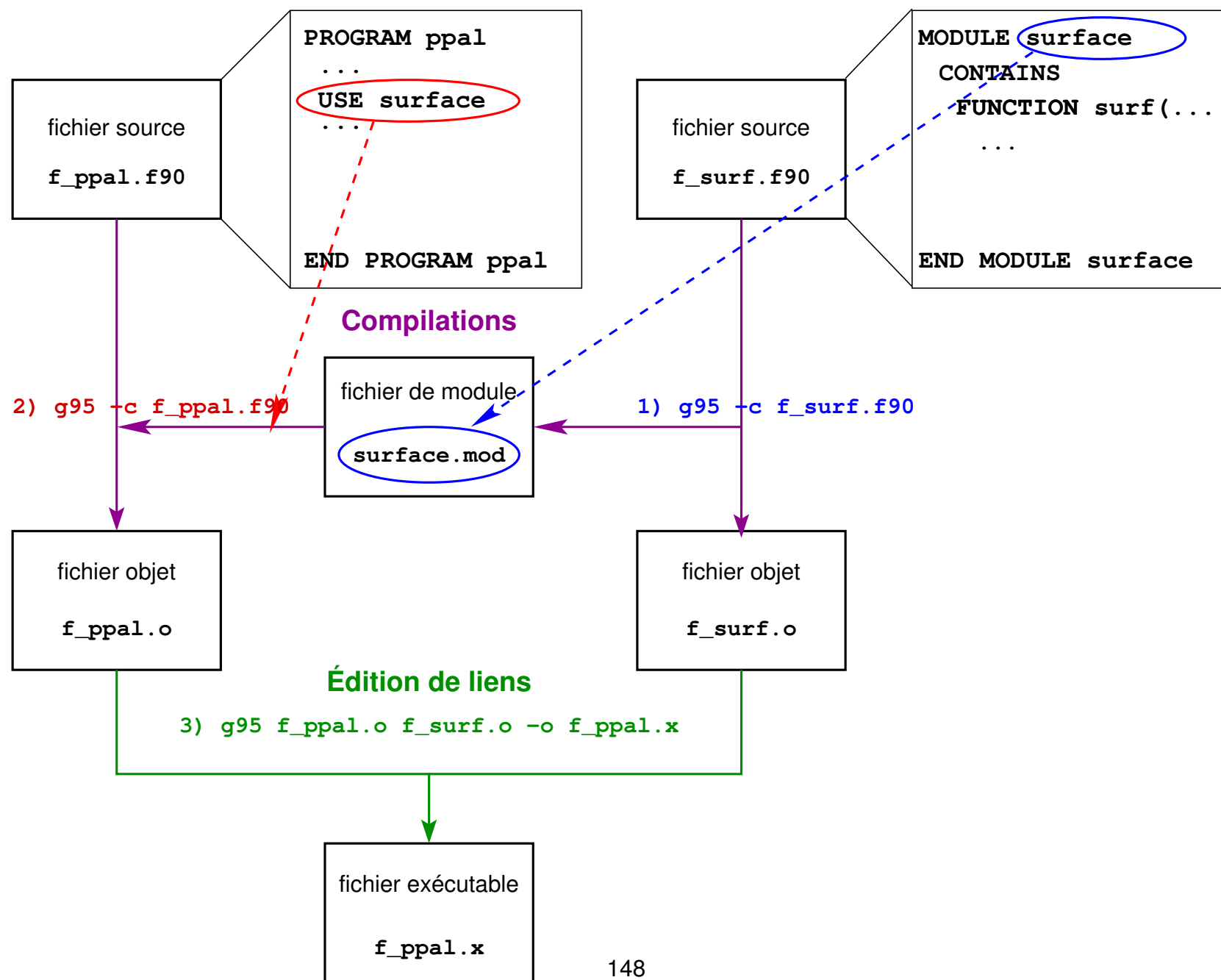
7.3.3 Exemples d'usage de variable globale

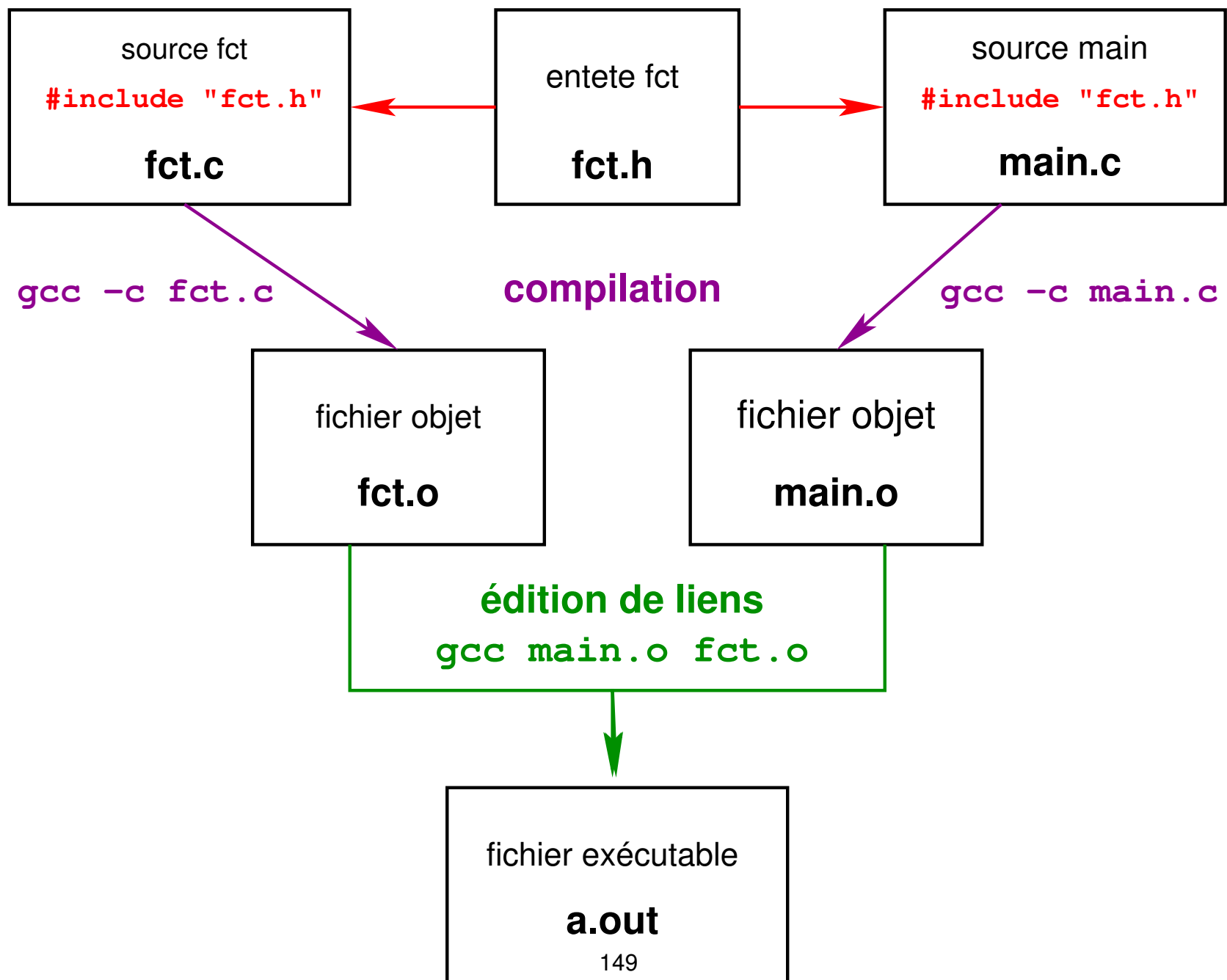
```
#include<stdio.h> /* fichier compte-globale2.c */
#include<stdlib.h>
/* n= variable globale pour communiquer entre la fct avance et le m
int n; /* variable globale déclarée avant avance */
void avance(void) ; /* déclaration de avance */
void avance(void) { /* définition de avance */
    n++; /* surtout ne pas redéclarer n ! */
    printf("%d \n", n); /* effet de bord */
    return;
}
int main(void) {
    int i;
    n = 0; /* globale mais ne pas redéclarer sinon masquage */
    for( i = 1; i<=5; i++) {
        avance(); /* aucun transfert de paramètre */
    }
    exit(0);
}
```

```
MODULE m_compte ! fichier compte-globale2.f90
IMPLICIT NONE
INTEGER :: n ! assure la communication
CONTAINS
  SUBROUTINE avance ! sous programme sans argument
    IMPLICIT NONE
    n = n + 1 ! ne pas redéclarer n sinon masquage !
    WRITE(*,*) n ! effet de bord
  END SUBROUTINE avance
END MODULE m_compte
PROGRAM t_compte
  USE m_compte ! assure la visibilité de n
  INTEGER :: i
  n = 0 ! redéclaration interdite ici
  DO i = 1, 5
    CALL avance ! appel du sous programme sans argument
  END DO
  STOP
END PROGRAM t_compte
```

7.4 Visibilité des interfaces et compilation séparée

Langage C	Fortran 90
	procédures internes introduites par CONTAINS ⇒ non réutilisables
assurer la visibilité des interfaces des procédures externes pour permettre les contrôles inter-procéduraux par le compilateur	
fonctions externes prototype de fonction dans chaque fichier source – soit explicitement dupliqué (risques) – soit dans un fichier d'entête incorporé par le préprocesseur dans le fichier de la fonction et celui de l'appelant via #include f_entete.h +protection contre inclusions multiples	procédures externes – soit déclaration explicite de l'interface dans l'appelant, mais duplication de code (risqué) – soit module et procédure "interne de module", plus instruction USE dans l'appelant (fiable)
compilation séparée	
unité de compilation : le fichier	unité de compilation : le module





7.5 Compléments sur les procédures

Langage C	Fortran 90
Récurtivité des procédures	
récurtivité autorisée pour les fonctions par défaut en C	déclarer l'attribut RECURSIVE et la variable résultat par RESULT (<code>var</code>) (fcts)
nb d'arguments variable (ex. <code>printf</code>) C++ arg. optionnels et valeurs par défaut	arguments optionnels attribut : optional test : if (present (var))
	arguments à mot-clef = le paramètre formel
en C99 fonctions mathématiques génériques entre variantes de type avec #include <tgmath.h> utilisation des pointeurs génériques <code>void *</code> pour manipuler plusieurs types de données (tri avec <code>qsort</code>)	procédures génériques : même appel pour différents types (suppose la présence de versions spécifiques pour chaque type) les fcts mathématiques intrinsèques sont génériques

7.5.1 Exemples de procédures récursives : factorielle

Factorielle récursive en C

```
int fact(int n){ /* calcul récursif de factorielle */  
/* attention aux dépassements de capacité non testés en entier */  
int factorielle; /* limiter à n<=12 */  
if ( n > 1 ){  
    factorielle = n * fact(n-1); /* provoque un autre appel à fact */  
}  
else {  
    factorielle = 1 ; /* arrêt de la récursion */  
}  
return factorielle;  
}
```

version idiomatique : `return n>1 ? n*fact(n-1) : 1;`


```
#include <stdio.h>          /* fichier t_fct_recur+save.c */
#include <stdlib.h>
int n_appels; /* variable globale */
int fact(int n);
int fact(int n) {          /* calcul récursif de factorielle */
    /* attention aux dépassements de capacité non testés en entier */
    int factorielle; /* ne pas utiliser au dela de 12! */
    n_appels++; /* incrémentation à chaque appel */
    if ( n > 1 ) {
        fprintf(stderr, "avant n = % *d \n", 3*(13-n), n); /* format variable */
        factorielle = n * fact(n-1); /* provoque un autre appel à fact */
        fprintf(stderr, "après n = % *d ", 3*(13-n), n); /* décalage de 3 esp.*/
        fprintf(stderr, " factor. = %d\n", factorielle);
    }
    else {
        factorielle = 1 ;
        fprintf(stderr, "          n = % *d arrêt empilage\n", 3*(13-n), n);
    }
    return factorielle;
}
```

```
/* factorielle récursive : attention aux dépassements de capacité */

int main(void) {
int i=1;
do{ /* boucle tant que i > 0 */
    printf(" entrer un entier (<=0 pour terminer)\n");
    scanf("%d", &i);
    n_appels = 0;      /* RAZ de la variable globale */
    printf("i = %d i! = %d\n", i, fact(i));
    printf(" nb appels (pour calcul de i!) %d\n", n_appels); /* affichage */
} while (i > 0);
return 0;
}
```

```
avant n =          9
avant n =         8
avant n =        7
avant n =        6
avant n =        5
avant n =        4
avant n =        3
avant n =        2
    n =          1 arrêt empilage
après n =          2 factor. = 2
après n =          3 factor. = 6
après n =          4 factor. = 24
après n =          5 factor. = 120
après n =          6 factor. = 720
après n =          7 factor. = 5040
après n =          8 factor. = 40320
après n =          9 factor. = 362880
i = 9 i! = 362880
nb appels (pour calcul de i!) 9
```

Fonction factorielle récursive en fortran

```
! fichier t_fct_rekurs.f90  
INTEGER RECURSIVE FUNCTION fact(n) RESULT(factorielle)  
! le type entier s'applique en fait au résultat : factorielle  
! attention aux dépassements de capacité en entier non testés !!  
IMPLICIT NONE  
INTEGER, INTENT(IN)      :: n  
IF ( n > 1 ) THEN  
    factorielle = n * fact(n-1) ! provoque un nouvel appel de fact  
ELSE  
    factorielle = 1 ! arrêt de la récursivité (nécessaire)  
END IF  
END FUNCTION fact
```

Version subroutine avec comptage des appels par variable statique locale

```

MODULE m_fact2 ! fichier t_recur+save.f90
IMPLICIT NONE
CONTAINS ! version subroutine + comptage du total des appels
RECURSIVE SUBROUTINE fact(n, factorielle)
  INTEGER, INTENT(IN) :: n
  INTEGER, INTENT(OUT) :: factorielle
  INTEGER, SAVE :: n_appels = 0 !statique => commun aux diff. appel
  n_appels = n_appels + 1
  IF ( n > 1 ) THEN
    CALL fact(n-1, factorielle) ! appel récursif
    factorielle = n * factorielle
  ELSE
    factorielle = 1 ! fin de la récursion
    WRITE(*,*) "nb total d'appels", n_appels ! affichage
    n_appels = 0 ! RAZ nécessaire sinon cumul
  END IF
END SUBROUTINE fact
END MODULE m_fact2

```

```
PROGRAM t_fact2  ! deux appels seulement pour simplifier
  ! le compteur d'appels n'est pas visible ici (RAZ impossible)
USE m_fact2
INTEGER :: m, p
WRITE (*,*) "entrer un entier m <= 12"
READ (*,*) m
CALL FACT(m, p)
WRITE (*,*) "m = ", m, " m! = ", p
WRITE (*,*) "entrer un entier m <= 12"
READ (*,*) m
CALL FACT(m, p)
WRITE (*,*) "m = ", m, " m! = ", p
END PROGRAM t_fact2
```

Initialisation (une fois à la compilation) et RAZ du compteur dans la subroutine car compteur = variable locale non visible de l'appelant

Version avec comptage des appels par variable de module

```
MODULE m_fact3 ! version subroutine + comptage du total des appels
IMPLICIT NONE ! fichier t_recur+glob.f90
INTEGER :: n_appels ! visible des procédures du module et via use
CONTAINS
RECURSIVE SUBROUTINE fact(n, factorielle)
  INTEGER, INTENT(IN)      :: n
  INTEGER, INTENT(OUT)    :: factorielle
  n_appels = n_appels + 1
  IF ( n > 1 ) THEN
    CALL fact(n-1, factorielle) ! appel récursif
    factorielle = n * factorielle
  ELSE
    factorielle = 1 ! fin de la récursion
  END IF
END SUBROUTINE fact
END MODULE m_fact3
```


```
PROGRAM t_fact3  ! deux appels seulement pour simplifier
  USE m_fact3   ! => visibilité de n_appels
  INTEGER :: m, p
  n_appels = 0   ! initialisation dans le pgm ppal; ne pas masquer
  WRITE(*,*) "entrer un entier m <= 12"
  READ (*,*) m
  CALL FACT(m, p)
  WRITE(*,*) "m = ", m, " m! = ", p
  WRITE(*,*) "nb total d'appels", n_appels ! affichage
  n_appels = 0   ! réinitialisation du compteur d'appels
  WRITE(*,*) "entrer un entier m <= 12"
  READ (*,*) m
  CALL FACT(m, p)
  WRITE(*,*) "m = ", m, " m! = ", p
  WRITE(*,*) "nb total d'appels", n_appels ! affichage
END PROGRAM t_fact3
```


7.5.2 Les fonctions mathématiques

Langage C : C89, C99	fortran
#include <math.h> pour le compilateur C89 -lm pour l'éditeur de lien (<code>libm.a</code>)	bibliothèque intrinsèque standard
Généricité des fonctions mathématiques	
#include <tgmath.h> pour le compilateur C99 appelle versions float et long double selon arg. ex : sin (générique) appelle <code>sinf</code> , <code>sin</code> ou <code>sinl</code>	par défaut
man 3 fct pour le manuel	

type des arguments dans les tableaux suivants

i	r	z
entier	flottant	complexe

fortran	C89	C99 avec <code>tgmath.h</code>	remarques
 ABS (n)	<code>l/l1/abs (n) (i)</code>		abs pour les entiers en C
ABS (a)	fabs /f/l (a) (r)	<code>fabs (a) (a)</code>	C99 + tgmath.h
ABS (c)		<code>cabs (c) (z)</code> C99 + fabs	complex.h
SQRT (a)	sqrt /f/l (a) (r)	<code>csqrt (a) (z)</code> sqrt	
a**b	pow /f/l (a,b) (r)	<code>cpow/f/l (a,b) (z)</code> pow	opérateur en fortran
EXP (a)	exp /f/l (a) (r)	<code>cexp/f/l (a) (z)</code> exp	
LOG (a)	log /f/l (a) (r)	<code>clog/f/l (a) (z)</code> log	
LOG10 (a)	log10 /f/l (a) (r)		
REAL (a)		creal /f/l (a) (z)	C99 + complex.h
AIMAG (a)		cimag /f/l (a) (z)	C99 + complex.h
CONJ (a)		conj /f/l (a) (z) conj	C99 + complex.h
CEILING (a)	ceil /f/l (a) (r)		résultat flottant en C
FLOOR (a)	floor /f/l (a) (r)		résultat flottant en C
NINT (a)	<code>l/lrint</code> /f/l (a) (r)		arrondi entier long en C
MOD (n,p)	<code>n%p (i, i)</code>		opérateur en C
MODULO (n,p)	<code>n%p (i, i)</code>		opérateur en C
SIGN (a,b)		copysign /f/l (a,b)	$ a \times \text{signe}(b)$

fortran	C89	C99 avec <code>tgmath.h</code>	remarques
COS (a)	cos /f/l (a) (r)	ccos/f/l (a) (z) cos	
COSH (a)	cosh /f/l (a) (r)	ccosh (a) (z) cosh	
ACOS (a)	acos /f/l (a) (r)	caacos/f/l (a) (z) acos	C99 + tgmath.h
SIN (a)	sin /f/l (a) (r)	csin/f/l (a) (z) sin	
SINH (a)	sinh /f/l (a) (r)	csinh/f/l (a) (z) sin	C99 + tgmath.h
ASIN (a)	asin /f/l (a) (r)	casin/f/l (a) (z) asin	
TAN (a)	tan /f/l (a) (r)	ctan/f/l (a) (z) tan	
TANH (a)	tanh /f/l (a) (r)	ctanh/f/l (a) (z) tanh	C99 + tgmath.h
ATAN (a)	atan /f/l (a) (r)	atan/f/l (a) (z) atan	
ATAN2 (a, b)	atan2 /f/l (a, b)	atan2	

fortran 2008	C89 (r)	C99 (z) avec <code>tgmath.h</code>	remarques
ACOSH (a)	acosh /f/l (a)	cacosh (a) acosh	C99 + tgmath.h
ASINH (a)	asinh /f/l (a)	casinh/f/l (a) asin	C99 + tgmath.h
ATANH (a)	atanh /f/l (a)	catanh/f/l (a) atanh	C99 + tgmath.h
BESSEL_J0 (a)	j0 /f/l (a)	les fonctions de Bessel j0, j1, jn, y0, y1 et yn ne sont pas standard en C, y compris dans les normes C99 ou C2011	$J_0(a)$
BESSEL_J1 (a)	j1 /f/l (a)		$J_1(a)$
BESSEL_JN (n, a)	jn /f/l (n, a)		$J_n(n, a)$
BESSEL_Y0 (a)	y0 /f/l (a)		$Y_0(a)$
BESSEL_Y1 (a)	y1 /f/l (a)		$Y_1(a)$
BESSEL_YN (n, a)	yn /f/l (n, a)		$Y_n(n, a)$
ERF (a)	erf /f/l (a)		$\frac{2}{\sqrt{\pi}} \int_0^a e^{-t^2} dt$
ERFC (a)	erfc /f/l (a)	$\frac{2}{\sqrt{\pi}} \int_a^\infty e^{-t^2} dt$	
GAMMA (a)	tgamma /f/l (a)		$\Gamma(a)$
LOG _GAMMA (a)	lgamma /f/l (a)		$\ln(\Gamma(a))$
HYPOT (a, b)	hypot /f/l (a, b)		$\sqrt{a^2 + b^2}$

7.5.3 Une erreur classique : la fonction `abs` en flottant en C

La fonction `abs` a pour prototype C89 : `int abs (int j) ;`

Si on lui passe un double, il est converti en entier avant prise de la valeur absolue.

`abs (.5)` donne `0`

⇒ penser à `fabs` de prototype : `double fabs (double x) ;`

Variantes : en C99, il existe trois variantes pour chacune de ces fonctions selon les 3 variantes de type flottant et les 3 variantes de type entier.

Ex. : `fabs` pour `double`, `fabsf` pour `float` et `fabsl` pour `long double`

Un `float` passé à `fabs` est converti en `double`, mais n'est pas converti s'il est passé à `fabsf`.

Généricité : avec `#include <tgmath.h>`, on écrit toujours `abs` et `fabs` et la fonction spécifique est choisie pour éviter le changement de variante. La généricité recouvre réels et complexes mais pas entiers et flottants.

8 Tableaux

8.1 Définition et usage

Un **tableau** est un ensemble «rectangulaire» d'éléments **de même type**, stockés de façon contigüe en mémoire (pas imposé en fortran) et repérés au moyen d' **indices entiers**.

L'ensemble de ces objets est identifié par un **identifiant unique** : le nom du tableau.

Les éléments d'un tableau sont rangés selon un ou plusieurs axes : les **dimensions** du tableau

En fortran, le nombre de dimensions est appelé le **rang** du tableau.

On représentera par exemple :

— un vecteur par un tableau à une dimension (rang 1)

N.B. : pas de différence entre vecteur-ligne et vecteur-colonne (\neq scilab/matlab)


— une matrice par un tableau à 2 dimensions (rang 2)...

étendue d'un tableau selon une dimension = le nombre d'éléments selon cet axe

profil d'un tableau = vecteur de ses étendues

taille d'un tableau = nombre total d'éléments = produit de ses étendues

8.2 Tableaux de dimension fixe

Langage C	fortran 90
Déclaration	
<code>int tab1[3] ;</code>	<code>INTEGER, DIMENSION(3) :: tab1</code>
<code>int tab2[3][2] ;</code>	<code>INTEGER, DIMENSION(3,2) :: tab2</code>
	<code>INTEGER :: tab2(3,2), tab1(3)</code>
Référence à un élément du tableau	
<code>i = tab1[2] ;</code>	<code>i = tab1(3)</code>
<code>i = tab2[2][0] ;</code>	<code>i = tab2(3,1)</code>
<p> opérateur séquentiel « , » \Rightarrow [2, 0] interprété comme [0] ! mais avertissement compilateur</p>	

Langage C	fortran 90
Indexation différente	
<p>⚠ commence à 0 ⇒ termine à N-1 possibilité de décaler l'indexation avec un pointeur</p>	<p>commence à 1 par défaut choix possible à la déclaration ex : <code>DIMENSION (-2 : 2)</code> ⇒ de -2 à +2</p>
Rangement différent ⇒ conséquence sur les performances	
indice rapide = qui défile le plus vite = celui des éléments contigus	
<p>⚠ le plus à droite = le dernier tabl. 2D = tableau de tableaux ⇒ matrices rangées par lignes</p>	<p>le plus à gauche = le premier ⇒ matrices rangées par colonnes</p>
Constructeurs de tableaux	
<pre>int tab1[3]={1, 2, 3};</pre>	<pre>INTEGER, DIMENSION(3)::tab1=(/1, 2, 3/) tab1=[1, 2, 3] en fortran 2003</pre>
possibilité d'imbriquer si dim > 1	ne pas imbriquer si dim > 1 => RESHAPE

8.3 Tableaux en fortran

Manipulation globale des tableaux selon une syntaxe proche de celle de matlab/scilab/octave (mais vecteur ligne = vecteur colonne = tableau 1D) :

- généralisation des opérateurs et des fonctions scalaires « élémentaires » aux tableaux **conformants** (même profil)
- promotion des scalaires dans les opérations avec les tableaux
- affectation globale **tab = 0**
- **fonctions spécifiques tableaux** : interrogation, réduction et transformation
- sections régulières de tableaux **TAB (début : fin : pas)**
sélection de ligne et de colonne (un des deux est impossible en C)
- instructions tableaux (**FORALL**, **WHERE**) et parallélisation

8.3.1 Opérations globales sur les tableaux en fortran

```

REAL, DIMENSION(10,3) :: mat1, mat2, mat3
mat1(:, :) = 2. * mat2(:, :) + 1. ! scalaires promus tableaux
mat3(:, :) = mat1(:, :) + mat3(:, :) ! si conformants
mat3(:, :) = COS(mat3(:, :)) ! fonction scalaire "élémentaire"
mat3(:, :) = mat3(:, :) * mat3(:, :) ! multipl. terme à terme

```

8.3.2 Sections régulières de tableaux en fortran

```

REAL, DIMENSION(3,6) :: mat ! 3 lignes x 6 col
REAL, DIMENSION(3) :: col ! vecteur, éviter (3,1)
REAL, DIMENSION(3,3) :: smat ! 3 lignes x 3 col
col(:) = mat(:,5) ! col = la 5ème col.
smat(:, :) = mat(:, 2:6:2) ! colon. 2, 4 et 6
col(:) = mat(2, 1:3) ! col = partie de la ligne 2

```

début : fin : pas en fortran mais  début : pas : fin en scilab/octave/matlab

matrice (1:2, 2:6:2)

	1	2	3	4	5	6
1		x		x		x
2		x		x		x
3						

matrice (:::2, 2:6:2)

	1	2	3	4	5	6
1		x		x		x
2						
3		x		x		x

8.3.3 Fonctions opérant sur des tableaux en fortran

```

REAL, DIMENSION (10, 3)      :: mat1, mat2, mat3
REAL, DIMENSION (3, 10)     :: mat4
REAL, DIMENSION (10, 10)    :: mat5
REAL, DIMENSION (100)       :: vect1, vect2
REAL                          :: rmax , rs, rmoy
INTEGER                       :: n1, n2, n
INTEGER, DIMENSION (2)      :: locmax , profil

```

```

profil = SHAPE (mat1)           ! vecteur des étendues (profil)
    nb d'éléments du profil = rang du tableau
n = SIZE (mat1)                 ! nombre total d'éléments
    argument optionnel DIM=
n1 = SIZE (mat1, 1) ; n2 = SIZE (mat1, DIM=2) ! étendues
rs = SUM (vect1)                ! somme
rmoy = SUM (vect1) / SIZE (vect1) ! moyenne (vect1 réel sinon REAL())
    SUM (mat1, DIM=2)  somme selon l'axe 2 (ligne par ligne)
rs = PRODUCT (mat1)             ! produit des éléments
    SIZE (mat1)  est aussi PRODUCT (SHAPE (mat1))
rmax = MAXVAL (mat1)           ! valeur maximale du tableau
locmax (:) = MAXLOC (mat2)      ! vecteur de la position du max
imax = MAXLOC (vect1, dim=1) ! indice de la position du max (partant de 1)
mat4 (:, :) = TRANSPOSE (mat1) ! matrice transposée
mat5 (:, :) = MATMUL (mat1, mat4) ! multiplication matricielle
rs = DOT_PRODUCT (vect1, vect2) ! produit scalaire

```

Utilisation des **masques** (tableaux booléens) : mot-clef **MASK**

SUM(vect1, **MASK=vect1>0**) somme des éléments positifs

LBOUND / **UBOUND** tableaux 1D des **bornes des indices**

(taille = le rang du tableau)

N.-B. : **LBOUND**(vecteur) est un **vecteur** de taille 1

mais **LBOUND**(vecteur, **DIM=1**) est un scalaire

RESHAPE(vect, shape) tableau 1D vect → tableau de profil shape

changement de profil utilisé pour initialiser un tableau de rang supérieur à 1

INTEGER, DIMENSION(2:3) :: mat=**RESHAPE**([1, 2, 3, 4, 5, 6], **[2, 3]**)

SPREAD étend par **duplication**

CSHIFT / **EOSHIFT** **décalages** circulaires/décalages avec pertes

PACK / **UNPACK** **extraction/ventilation** selon des masques

MERGE **fusion** de tableaux selon un masque

8.3.4 Parallélisation en fortran

- **ALL/ANY** (*masque logique*) ET/OU logique sur un tableau de booléens
 - IF ALL (v > 0) v = log(v)** ! si tous les élts de v sont > 0
 - IF ANY (v < 0) v = v-minval(v)** ! si au moins 1 des élts de v est < 0
 - n = COUNT(v>0)** ! nombre des éléments positifs
- **FORALL (i = 1:size(v)) w(i, i) = v(i)** ! vecteur v -> diagonale de w
 - FORALL (i = 1: size(v) -1)**
 - v(i + 1) = v(i+1) - v(i)** ! calcul de "gradient" quel que soit l'ordre
 - END FORALL**
- **WHERE (v > 0)**
 - v = log(v)** ! log protégé
 - ELSEWHERE**
 - v = -1.e20**
 - ENDWHERE**

8.3.5 Ordre des éléments de tableaux 2D en fortran

```
PROGRAM tab2d ! tab2d.f90
! impression d'un tableau à 2 dimensions
IMPLICIT NONE
INTEGER, PARAMETER :: lignes=3, colonnes=4
INTEGER, DIMENSION(lignes,colonnes) :: t
INTEGER :: i
t(1,:) = (/11, 12, 13, 14/) ! [11, 12, 13, 14] en f2003
t(2,:) = (/21, 22, 23, 24/)
t(3,:) = (/31, 32, 33, 34/)
WRITE(*,*) "impression du tableau 2d t(i,j)=10i+j"
WRITE(*,*) "en faisant varier j à i fixé"
DO i = 1, lignes
    WRITE(*,*) t(i, :) ! j=1 à 4 ...
END DO
```

```
WRITE (*, *)  
WRITE (*, *) "impression globale du tableau 2d t(i,j)=10i+j"  
WRITE (*, *) "en suivant l'ordre en mémoire"  
! impression globale du tableau (pas de retour à la ligne)  
WRITE (*, *) t(:, :)  
WRITE (*, *) "=> l'indice le plus à gauche varie le plus vite"  
END PROGRAM tab2d
```


impression du tableau 2d $t(i, j) = 10i + j$

en faisant varier j à i fixé

11 12 13 14

21 22 23 24

31 32 33 34

impression globale du tableau 2d $t(i, j) = 10i + j$

en suivant l'ordre en mémoire


11 21 31 12 22 32 13 23 33 14 24 34

⇒ l'indice **le plus à gauche** varie le plus vite (rangement par colonnes)

8.4 Tableaux et pointeurs et en C

En langage C, tout **identificateur de tableau** apparaissant dans une expression est converti en une **constante** de type **pointeur vers le type des éléments** du tableau dont la valeur est l'**adresse du premier élément** :

```
float tab[9] ;    ⇒ tab est converti en un pointeur vers le float tab[0]
                  ⇒ tab = &tab[0] ⇒ tab[0] = *tab
float f ; f = tab[8] ; (dernier élément)
```

 L'affectation globale `tab =` est donc **impossible**. Mais si on déclare un pointeur vers un réel `float *pf ;`, on peut écrire `pf = tab ;`.

Arithmétique pointeur : la somme `tab+i` d'un entier `i` et du pointeur `tab` est interprétée comme un pointeur contenant l'adresse de l'élément d'indice `i` du tableau, `tab[i] = *(tab+i)` = i[tab] (!)

Par exemple, `pf` sous-tableau commençant au 4^e élément de `tab` :

```
float *pf ;    pf = &tab[3] ; ou aussi pf = tab + 3 ;
```

8.4.1 Ordre des éléments de tableaux 2D en C

```
#include <stdio.h>                                     /* tab2d.c */
#include <stdlib.h>
#define LIGNES 3
#define COLONNES 5

/* impression d'un tableau à 2 dimensions */
int main(void) {
    int t [LIGNES] [COLONNES] = { {11,12,13,14,15},
                                   {21,22,23,24,25},
                                   {31,32,33,34,35}
                                   } ;

    int i, j;
    int *k = &t[0][0]; /* k est un pointeur d'entier */
    printf("impression du tableau 2d t[i][j]=10(i+1)+(j+1)\n");
    printf("en faisant varier j à i fixé\n");
```

```
for (i = 0 ; i < LIGNES; i++) {           /* indice lent */
    for (j = 0 ; j < COLONNES; j++) {     /* indice rapide */
        printf("%d ", t[i][j]) ;
    }
    printf("\n") ;
}
printf("impression du tableau 2d t[i][j]=10(i+1)+(j+1)\n");
printf("en suivant l'ordre en mémoire\n");
for (i = 0 ; i < LIGNES * COLONNES; i++) {
    printf("%d ", *(k+i)) ;
}
printf("\n") ;
printf("=> l'indice le plus à droite varie le plus vite\n");
exit(EXIT_SUCCESS) ;
}
```

impression du tableau 2d $t[i][j] = 10(i+1) + (j+1)$

en faisant varier j à i fixé

11 12 13 14 15

21 22 23 24 25

31 32 33 34 35

impression du tableau 2d $t[i][j] = 10(i+1) + (j+1)$

en suivant l'ordre en mémoire

11 12 13 14 15 21 22 23 24 25 31 32 33 34 35

⇒ l'indice **le plus à droite** varie le plus vite (rangement par lignes)

En C, pas de vrais tableaux 2D, mais des **tableaux de tableaux**.

8.4.2 Sous-tableau 1D avec un pointeur

```
#include <stdio.h> /* sous-tab1d.c */
#include <stdlib.h>
#define N 10

/* manipulation d'un sous-tableau 1d avec un pointeur */
int main(void) {
double tab[N] ; /* tableau initial */
double *ptrd=NULL; /* pointeur sur le même type que les éléments */
int i ;

for (i = 0 ; i < N ; i++) {
    tab[i] = (double) i ; /* remplissage du tableau */
}
```

```
ptrd = tab + 3; /* arithm.pointeurs: équivaut à ptrd=&tab[3]

/* affichage du tableau et du sous tableau */
for (i = 0 ; i < N ; i++)
{
    printf(" tab[%d] = %f", i, tab[i]);
    if (i < N - 3 ) { /* au delà, sortie du tableau initial */
        printf(" *(ptrd+%d) = %f", i, *(ptrd+i));
        printf(" ptrd[%d] = %f", i, ptrd[i]);
    }
    printf("\n") ;
}
exit (EXIT_SUCCESS) ;
}
```

8.4.3 Utilisation de `typedef`

`typedef` permet de définir des synonymes de types en C

Recette syntaxique : dans une déclaration classique, remplacer le nom de la variable par **le synonyme** et insérer `typedef` en tête

Exemple 1 : choisir les types flottants de façon paramétrée

```
typedef float real; ou typedef double real;
```

puis,

```
real x, y;
```

Exemple 2 : syntaxe plus délicate avec les tableaux

```
typedef float vect[3];
```

puis


```
vect u, v;
```

2 tableaux de 3 float

```
vect tv[100];
```

tv tableau de 100 tableaux de 3 float
équivalent à `float tv[100][3];`

8.5 Procédures et tableaux

Langage C	Fortran 90
tableau = pointeur \Rightarrow valeurs modifiables (sauf si const aux valeurs pointées)	INTENT (OUT) , INTENT (INOUT) INTENT (IN)
passage de tableaux de dimensions fixes (pour le compilateur) en argument \Rightarrow pas de problème particulier, mais paramétrer les dimensions ! #define N	attribut PARAMETER
Tableau 1D de dim inconnue à la compilation de la procédure :  transmettre le nombre d'éléments à la fct \Rightarrow notation <code>t [i]</code> utilisable	ne pas transmettre le nb d'éléments \Rightarrow déclarer DIMENSION (:) dans la procédure et utiliser SIZE si besoin
Tableau 2D ou plus de dim inconnue à la compilation : utiliser des pointeurs , transmettre le nombre d'éléments et calculer les adresses C99 : tableaux automatiques de taille variable transmettre les dimensions avant	déclarer le rang DIMENSION (:, :) dans la procédure et utiliser SIZE (... , DIM=...) si besoin

8.5.1 Passage d'un tableau 1D en fortran

```

! passage en argument d'un tableau 1d de taille variable
MODULE m_fct1d                                ! fichier tab1d+fct2.f90
IMPLICIT NONE
CONTAINS
  SUBROUTINE double_tab(tt)
    INTEGER, DIMENSION(:), INTENT(INOUT) :: tt
    tt(:) = 2 * tt(:) ! doublement des valeurs
  END SUBROUTINE double_tab
  SUBROUTINE print_tab(tt)
    INTEGER, DIMENSION(:), INTENT(IN) :: tt
    INTEGER :: i
    WRITE(*, *) tt(:) ! d'abord écriture en ligne
    DO i = 1, SIZE(tt) ! taille donnée par SIZE
      WRITE(*,*) tt(i) ! puis écriture en colonne
    END DO
  END SUBROUTINE print_tab
END MODULE m_fct1d

```

```
PROGRAM tab1d_fct          ! fichier tab1d+fct2.f90
USE m_fct1d
INTEGER, PARAMETER       :: n=5, p=2 ! constantes nommées
INTEGER, DIMENSION(n)   :: t
INTEGER, DIMENSION(p)   :: u
INTEGER                  :: i

DO i = 1, n
    t(i) = i
END DO

DO i = 1, p
    u(i) = 3 * i
END DO

CALL print_tab(t)        ! déconseillé de passer la taille
CALL double_tab(t)      ! du tableau à la procédure
CALL print_tab(t)
CALL print_tab(u)       ! appel avec une autre taille
END PROGRAM tab1d_fct
```

8.5.2 Passage d'un tableau 2D en fortran

```
! passage en argument d'un tableau 2d de taille variable
MODULE m_fct2d ! fichier tab2d+fct2.f90
IMPLICIT NONE
CONTAINS
  SUBROUTINE print_tab2(tt)
    ! n'indiquer que le rang (2 ici), pas le profil
    INTEGER, DIMENSION(:, :), INTENT(IN) :: tt
    INTEGER :: i
    ! récupération des étendues via SIZE
    WRITE(*,*) "tableau ", SIZE(tt,1), " x ", SIZE(tt,2)
    DO I=1, SIZE(tt,1)
      PRINT *, tt(i,:)
    END DO
  END SUBROUTINE print_tab2
END MODULE m_fct2d
```

```
PROGRAM tab2d_fct           ! fichier tab2d+fct2.f90
USE m_fct2d
INTEGER, DIMENSION(2,4) :: t
INTEGER, DIMENSION(3,5) :: u
INTEGER                  :: i, j
DO i = 1, 2
  DO j = 1, 4
    t(i, j) = 70 + 10*i + j
  END DO
END DO
DO i = 1, 3
  DO j = 1, 5
    u(i, j) = 10*i + j
  END DO
END DO
CALL print_tab2(t)        ! déconseillé de passer le profil
CALL print_tab2(u)        ! appel avec un autre profil
END PROGRAM tab2d_fct
```

8.5.3 Passage d'un tableau 1D en C

```
#include <stdio.h>          /* fichier tab1d+fct2.c */
#include <stdlib.h>
#define N 5 /* constantes définies par le préprocesseur */
#define P 3 /* car le qualificatif const est insuffisant */
/* passage en argument d'un tableau 1d de taille variable */
void double_tab(int tt[], const int nb) ;
void double_tab(int tt[], const int nb) {
    int i ;                /* ^^^^ non modifiable localement */
    for (i = 0 ; i < nb; i++){
        tt[i] *= 2 ;      /* tt[i] est modifiable car tt = pointeur */
    }
}
void print_tab(const int tt[], const int nb) ;
void print_tab(const int tt[], const int nb) {
    int i ; /* ^^^^^ tt[i] et ^^^^ nb non modifiables ds la fonct
```

```
printf(" impression du tableau de %d éléments\n", nb);
for (i = 0 ; i < nb; i++) {
    printf("%d ", tt[i]) ;           /* impression du tableau en ligne */
}
printf("\n") ;
}
int main(void) {
    int t[N], u[P] ;
    int i ;
    for (i = 0 ; i < N ; i++) { t[i] = i ; }
    for (i = 0 ; i < P ; i++) { u[i] = 3 * i ; }
    print_tab(t, N) ;
    double_tab(t, N) ;
    print_tab(t, N) ;
    print_tab(u, P) ;
    exit(EXIT_SUCCESS) ;
}                                     /* fichier tab1d+fct2.c */
```

8.5.4 Passage d'un tableau 2D en C (norme C89)

Vrai tableau de tableaux \Rightarrow 2^e dimension fixe

```
#include <stdio.h>                                /* fichier tab2d+fct.c */
#include <stdlib.h>
#define P 5
/* transmission d'un tableau de rang > 1 à une fonction */
/*seule la dimension 1 peut être variable (passée en argument)*/
/* pour pouvoir utiliser la notation tableau de tableaux */
void print_mat(const int (* tt) [P], const int n) ;
void print_mat(const int (*tt) [P], const int n) {
/* void print_mat(const int tt[][P], const int n) {*/
    int i, j ;
    printf(" impression d'un tableau %d x %d\n", n, P);
    for (i = 0 ; i < n; i++) {
        for (j = 0 ; j < P; j++) {
            printf("%d ", tt[i][j]) ;
        }
    }
}
```



```
        printf("\n") ;
    }
}
int main(void) {
int t[3][P] = { {11,12,13,14,15},
                {21,22,23,24,25},
                {31,32,33,34,35} } ;
int u[2][P] = { {91,92,93,94,95},
                {81,82,83,84,85} } ;
print_mat((const int (*) [P]) t, 3) ;
printf("\n") ;
print_mat(u, 2) ;
printf("Concl : seule la première dimension peut être variable\n");
printf("pour que le compilateur puisse calculer l'adresse des éléments\n");
exit(EXIT_SUCCESS) ;
}                                     /* fichier tab2d+fct.c */
```

Simulation 2D avec calcul d'adresse sur tableau 1D

```
#include <stdio.h>                                /* fichier tab2d+fct2.c */
#include <stdlib.h>
/* transmission d'un tableau de rang 2 de taille variable à */
/* une fonction : pointeur et calcul explicite des adresses */
/* tableau déclaré de rang 1 dans la fonction          */
void print_mat(const int *tt, const int n, const int p) ;
void print_mat(const int *tt, const int n, const int p) {
    int i, j ;
    printf(" impression d'un tableau %d x %d\n", n, p);
    for (i = 0 ; i < n; i++) {
        for (j = 0 ; j < p; j++) { /* RowMajor en C */
            printf("%d ", *(tt + i*p + j) ); /* position i*p +j */
        }
        printf("\n") ;
    }
}
```

```
}

int main(void) {
int t[3][5] = { {11,12,13,14,15},
               {21,22,23,24,25},
               {31,32,33,34,35}
               } ;
int u[2][4] = { {91,92,93,94},
               {81,82,83,84}
               } ;
print_mat (&t[0][0], 3, 5); /* passer l'adresse du premier élément */
printf("\n") ;
print_mat (u[0], 2, 4) ; /* autre formulation de &u[0][0] */
printf("Concl : si plusieurs dimensions sont variables\n");
printf("le programmeur doit calculer l'adresse des éléments\n");
exit (EXIT_SUCCESS) ;
} /* fichier tab2d+fct2.c */
```

8.5.5 Passage d'un tableau 2D automatique de dimensions variables en C99

Passer le nombre d'éléments **avant** le tableau

 Le tableau **sans taille** dans l'appel, mais **avec taille** dans la définition


```
#include <stdio.h>                // fichier C99/fct+tab-var+size-c99.c
#include <stdlib.h>
// attention : ----- norme C99 -----
// passage en argument d'un tableau 1d de taille variable
// => il faut aussi passer la taille du tableau à la fct
// => il ft déclarer la taille du tabl. avant le tableau
void print_tab1(const int nb, const int tt[nb]) {
    int i ; // ^^^^^ nb et ^^^^^ tt non modifiables dans la fonction
    printf("  impression du tableau de %d éléments\n", nb);
    for (i = 0 ; i < nb; i++) {
        printf("%d ", tt[i]) ; // impression d'un élément du tableau
    }
    printf("\n") ;
}
```

```
// fonction à paramètres tableaux 2D de dim variable: C99 seulement
void print_tab2(const int n1, const int n2 , int tt[n1][n2]) {
    printf("impression du tableau de %d x %d éléments\n", n1, n2);
    for (int i = 0 ; i < n1; i++) {
        print_tab1(n2, tt[i]) ;    // impression du tableau 1D
    }
    printf("\n") ;
}

void tab_var(const int nn){ // C99 seulement
    // création de tableaux automatiques locaux de dim variable
    int ti[nn];           // 1D automatique sur la pile sans allocation
    int ti2[nn][2*nn];   // 2D automatique sur la pile sans allocation
    for (int i=0; i<nn; i++){
        ti[i] = i;
        for (int j=0; j<2*nn; j++){
            ti2[i][j] = 100*i+j ;
        }
    }
}
```

```
    }  
    print_tab1(nn, ti);           // affichage 1 D  
    print_tab2(nn, 2*nn, ti2);   // affichage 2 D  
    printf(" tailles en octets d'un scalaire: ti[0] %d\n", sizeof(ti[0]));  
    printf(" tailles en octets de ti (1D) %d et ti2 (2D) %d \n",  
           sizeof(ti), sizeof(ti2));  
    printf(" dimensions de ti2 (2D) %d %d \n", // calcul des tailles  
           sizeof(ti2)/sizeof(ti2[0]), sizeof(ti2[0])/sizeof(ti2[0][0]));  
    return ;  
}  
int main(void) {  
    int n ;  
    printf("entrer n ");  
    scanf("%d", &n);  
    tab_var(n); // tableaux non visibles dans le main  
    exit(EXIT_SUCCESS) ;  
} // fichier C99/fct+tab-var+size-c99.c
```

8.5.6 Tableaux automatiques locaux : C99 et fortran

	Langage C99	Fortran 90
	Alloués sur la « pile » (stack) \Rightarrow Portée et durée de vie limitées à la procédure pouvant être transmis aux procédures appelées , mais pas transmissibles à l'appelant désalloués automatiquement dès la sortie du scope (portée)	
	déclarer la taille avant le tableau et passer la taille en argument définition f(int n, int t[n]) \neq appel : f(n, t)	ne pas passer la taille en argument visibilité de l'interface : module + USE
	récupération de la taille (1 ^{re} dim.)	
	<code>sizeof(tab) / sizeof(tab[0])</code>	<code>SIZE(tab, DIM=1)</code>

Si la portée doit s'étendre à l'appelant (C89-99 et Fortran **2003**),

il faut gérer explicitement l'**allocation et la libération de la mémoire**

\Rightarrow **allocation dynamique** sur le « tas » (heap)

9 Allocation dynamique

9.1 Introduction

9.1.1 Trois types de tableaux

- **tableaux statiques** : occupent un emplacement défini avant l'exécution
- **tableaux automatiques** : pas d'emplacement défini avant exécution
 - ⇒ alloués et libérés «automatiquement», sur la **pile** (*stack*)
 - portée limitée (jusqu'à la fin du bloc en C99, de la procédure en fortran)
- **tableaux dynamiques** : pas d'emplacement défini avant exécution
 - ⇒ alloués et libérés «manuellement», sur le **tas** (*heap*)

Avantages des tableaux dynamiques :

- leur **taille** peut être choisie n'importe où lors de l'exécution du programme
- les tableaux dynamiques peuvent être alloués dans une procédure et rendus **accessibles dans l'appelant** (**portée** non limitée)

9.1.2 Cycle élémentaire d'un tableau dynamique

C'est au programmeur de se charger de l'allocation/libération dynamique de mémoire sur le tas :

1. choix de la taille du tableau
2. **allocation** : réussie ? (sinon fin)
3. utilisation du tableau
4. **libération** de la mémoire

⇒ **cycle de base** qui peut être itéré.



Ne pas oublier la libération (même en fin de programme principal) : elle peut être l'occasion de détecter une corruption de la zone allouée.

C : pointeurs obligatoires	fortran 90 : pointeurs possibles
allocation de mémoire	
<pre>void *malloc(size_t size) void *calloc(size_t nmemb, size_t size) initialise à 0</pre>	<pre>ALLOCATE (Objet (profil), STAT=err) où Objet déclaré ALLOCATABLE ou POINTER</pre>
en cas d'échec de l'allocation	
pointeur NULL	STAT /= 0
libération de la mémoire allouée	
<pre>void free (void *ptr) ajouter ptr=NULL; pour éviter erreur si autre free (ptr)</pre> <p style="text-align: center;">annuler aussi les pointeurs qui partagent cette cible</p>	<pre>DEALLOCATE (Objet, STAT=ierr) puis si pointeur, Objet => NULL</pre>
interrogation sur l'allocation/association	
	<pre>ALLOCATED(Objet allouable) → booléen ASSOCIATED(Objet pointeur) → booléen</pre>

9.1.3 Allocation dynamique en C avec `malloc` ou `calloc`

Deux fonctions standard pour allouer un espace mémoire **contigu** sur le tas.

Leur prototype est dans le fichier : `stdlib.h`

Prototype de `malloc` : `void *malloc(size_t taille) ;`

- Un argument `taille` (de type `size_t`), nombre d'octets à allouer :
⇒ utiliser `sizeof` qui donne la taille d'un élément
- Une valeur de retour du type `void *` (**pointeur générique** sur `void`) :
 - l'adresse de l'emplacement alloué si tout se passe bien,
⇒ **à convertir explicitement** en pointeur sur le type choisi
 - le pointeur `NULL` en cas de problème.
⇒ **à tester impérativement** avant d'utiliser la zone

Exemple : allocation d'un tableau 1D de 10 doubles

```
double *ptr = NULL ;
```

```
ptr = (double *) malloc(10*sizeof(double)) ;
```

Noter : conversion de `void*` en `double*` et utilisation de `sizeof`

Si `ptr != NULL`, le pointeur peut être ensuite utilisé avec le formalisme tableau :

```
ptr[0] soit *ptr, ... ptr[9] soit *(ptr+9)
```

Prototype de `calloc` :

```
void *calloc (size_t nb_blocs, size_t taille) ;
```

- Deux arguments spécifient le nombre d'octets à allouer :
 - `nb_blocs` : nombre de blocs consécutifs de `taille` octets à allouer,
 - `taille` : nombre d'octets par bloc (utiliser `sizeof`).
- Une valeur de retour du type `void *` (**pointeur générique** sur `void`) :
 - l'adresse de l'emplacement alloué si tout se passe bien,
 - le pointeur `NULL` en cas de problème.

`calloc` initialise tous les octets alloués à **zéro binaire**

(OK pour les entiers, problème possible pour les réels)

Exemple : allocation d'un tableau 1D de 10 doubles

```
double *ptr = NULL ;  
ptr = (double *) calloc(10, sizeof(double)) ;  
if (ptr == NULL) { fprintf(stderr, "erreur alloc\n");  
                exit(EXIT_FAILURE) ;  
}
```

9.1.4 Libération de la mémoire allouée en C avec `free`

La fonction standard `free` permet de libérer la mémoire allouée dynamiquement par `malloc` ou `calloc` (son prototype est dans le fichier : `stdlib.h`)

Prototype de `free` : `void free(void *adr) ;`

- Un argument `adr` de type pointeur générique :
 - ⇒ lui passer un pointeur contenant l'**adresse** de l'emplacement à libérer, adresse qui aura été fournie auparavant par `malloc` ou `calloc`
- Aucune valeur de retour.

Exemple : allocation puis libération d'un tableau 1D de 10 doubles

```
double *ptr = NULL ;
```

```
ptr = (double *) calloc(10, sizeof(double)) ;
```

```
...                                     travail sur le tableau alloué
```

```
free(ptr) ;
```

```
ptr = NULL ;                             pour plus de sécurité, en particulier si autre free
```

9.2 Allocation d'un tableau 1D

9.2.1 Allocation d'un tableau 1D en fortran

```

PROGRAM alloc1 ! fichier alloc-tab1d.f90
IMPLICIT NONE ! allocation dynamique sans pointeur
INTEGER :: i, n, ok=0
INTEGER, DIMENSION(:), ALLOCATABLE :: ti
    ! rang seulement ^^^^^^^^^ attribut obligatoire
DO ! boucle sur la taille du tableau jusqu'à n<=0
WRITE(*, *) "entrer le nb d'éléments du tableau (0 si fin)"
READ(*,*) n
IF (n <=0 ) EXIT ! sortie de la boucle
ALLOCATE(ti(n), stat=ok) ! allocation de la mémoire
IF (ok /= 0) THEN
    WRITE(*, *) "erreur d'allocation"
    CYCLE ! on passe à une autre valeur de n
END IF

```

```
DO i=1, n           ! affectation du tableau
  ti(i) = i
END DO
DO i=1, n           ! affichage du tableau en colonne
  WRITE(*,*) ti(i)
END DO
IF (ALLOCATED(ti)) THEN
  DEALLOCATE(ti) ! libération de la mémoire
END IF
END DO
END PROGRAM alloc1 ! fichier alloc-tab1d.f90
```

9.2.2 Allocation d'un tableau 1D en C

```
#include <stdio.h>
#include <stdlib.h>
int main(void) { /* fichier alloc-tab1d.c */
  int i, n;
```

```
int * pti = NULL; /* initialisation à NULL */
while( /* boucle globale sur la taille du tableau jusqu'à n<=0 */
    printf("entrer le nb d'éléments du tableau (0 si fin)\n"),
    scanf("%d", &n),
    n > 0 ) {
    pti = (int*) calloc((size_t) n, sizeof(int)); /* allocat. de n int */
    if (pti == NULL) {
        fprintf(stderr, "erreur d'allocation\n");
        continue; /* retour au choix de n sans utiliser le tableau */
    }
    for (i=0; i<n; i++) { /* affectation du tableau */
        pti[i] = i + 1 ;
    }
    for (i=0; i<n; i++) { /* affichage du tableau */
        printf("%d\n", pti[i]);
    }
    free((void*) pti) ; /* libération de la mémoire */
    pti = NULL ; /* par précaution si autre free(pti) */
}
exit(EXIT_SUCCESS) ;
} /* fichier alloc-tab1d.c */
```


9.3 Risques de fuite de mémoire

Si on alloue via un pointeur de tableau une cible anonyme : **pointeur = seul accès**

⇒ ne pas désassocier ce pointeur avant de libérer la zone

sinon zone mémoire **réservée mais inaccessible**

 **fuite de mémoire** (*memory leak*) ⇒ grave si dans une boucle

9.3.1 Fuite de mémoire avec les pointeurs en fortran

```
PROGRAM fuite_alloc_tab_ptr  ! fuite_alloc_tab_ptr.f90
IMPLICIT NONE
REAL, DIMENSION (:), POINTER :: ptr => NULL()
ALLOCATE(ptr(10))  ! allocation d'une cible anonyme de 10 réels
WRITE(*, *) ASSOCIATED(ptr)  ! affichera .true.
ptr(:) = 2  ! utilisation de la mémoire allouée
ptr => NULL()  ! désassociation avant déallocation ! => memory leak
WRITE(*, *) ASSOCIATED(ptr)  ! affichera .false.
END PROGRAM fuite_alloc_tab_ptr
```

Message à l'exécution avec g95 :

Remaining memory: 40 bytes allocated at line 4

⇒ aux pointeurs, préférer les tableaux allouables si possible

9.3.2 Fuite de mémoire en C

Ne pas **réaffecter le pointeur** conservant l'adresse d'une zone allouée avant de libérer par `free` la mémoire allouée (sauf si un autre pointeur permet d'accéder à la zone !)

```
#include <stdlib.h>
#include <stdio.h>
int main(void) {
float * ptr = NULL;
int n;
n = 10;
ptr = (float *) calloc((size_t) n, sizeof(float));
/* allocation d'une cible anonyme de 10 float */
ptr[9] = 9.; /* utilisation de la mémoire allouée */
/* nouvelle affectation de ptr par exemple */
/* ptr = (float *) calloc((size_t) 2*n, sizeof(float)); */
ptr = NULL; /* ou désassociation avant déallocation ! : */
/* zone réservée mais inaccessible => memory leak */
exit(EXIT_SUCCESS);
}
```

9.4 Application : manipulation de matrices

9.4.1 Matrices de taille quelconque en fortran

```
! lecture dans des fichiers de 2 matrices d'entiers  
! A(n, p) et B(p, m) avec  
! n, p et m quelconques donnés en première ligne des fichiers  
! => allocation dynamique  
! puis multiplication des matrices  
MODULE m_prod ! fichier produit_matr1.f90  
IMPLICIT NONE  
CONTAINS  
! la fonction intrinsèque matmul est bien sûr plus efficace  
SUBROUTINE prod_mat(a, b, c)  
  INTEGER, DIMENSION(:, :), INTENT(in) :: a, b ! entrée  
  INTEGER, DIMENSION(:, :), INTENT(out) :: c ! sortie  
  INTEGER :: n, p, m, i, j, k  
  n = SIZE(a, 1) ! récupération des dimensions  
  p = SIZE(a, 2)  
  m = SIZE(b, 2)
```

```

DO i = 1, n
  DO j = 1, m
    ! on pourrait se contenter de la ligne suivante
    ! c(i, j) = sum(a(i, :) * b(:, j)) ! produit "scalaire"
    c(i, j) = 0
    DO k = 1, p
      c(i, j) = c(i, j) + a(i, k) * b(k, j)
    ENDDO
  ENDDO
ENDDO
END SUBROUTINE prod_mat
END MODULE m_prod
!
PROGRAM produit_matrices
USE m_prod
INTEGER, PARAMETER :: unita = 10, unitb = 11
INTEGER :: n, m, p, pp
INTEGER, DIMENSION(:, :), ALLOCATABLE :: a, b, c
INTEGER :: i, j
INTEGER :: erreur_alloc

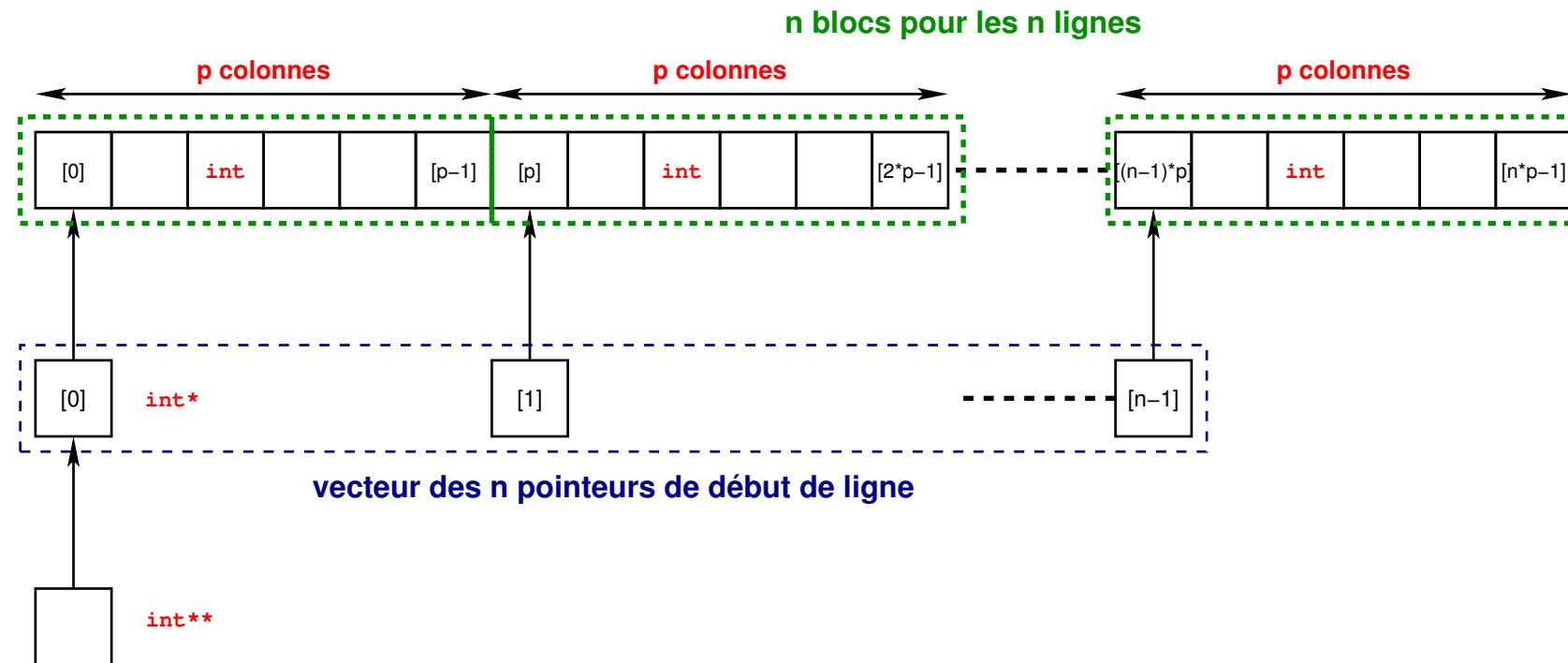
```

```
! lecture des dimensions des matrices
OPEN(unit=unita, file='mat-a.dat', form='formatted')
READ(unita, *) n, p
OPEN(unit=unitb, file='mat-b.dat', form='formatted')
READ(unitb, *) pp, m
IF (p /= pp) STOP 'matrices incompatibles'
! allocation des tableaux pour stocker ces matrices
ALLOCATE(a(n, p), b(p,m), c(n, m), stat = erreur_alloc)
IF( erreur_alloc /= 0 ) STOP ' erreur d''allocation '
! lecture des matrices dans les fichiers
DO i = 1, n ! un ordre par ligne
    READ (unita, *) a(i, 1:p)
ENDDO
CLOSE(unita)
DO i = 1, p ! un ordre par ligne
    READ (unitb, *) b(i, 1:m)
ENDDO
CLOSE(unitb)
```

```
! impression des matrices lues (par ligne)
WRITE(*,*) ' A ', n, ' x ', p
DO i = 1, n
    WRITE(*,*) a(i, 1:p)
ENDDO
WRITE(*,*) ' B ', p, ' x ', m
DO j = 1, p
    WRITE(*,*) b(j, 1:m)
ENDDO
! calcul du produit A.B : c = matmul(a, b) suffirait !
CALL prod_mat(a, b, c) ! méthode détaillée
! impression du résultat (par lignes)
WRITE(*,*) ' C = A * B : (' , n, ' x ', m, ') '
DO i = 1, n
    WRITE(*,*) c(i, :)
ENDDO
DEALLOCATE(a, b, c) ! libération des tableaux alloués
END PROGRAM produit_matrices ! fichier produit_matr1.f90
```

9.4.2 Matrices de taille quelconque en C : allocation en bloc

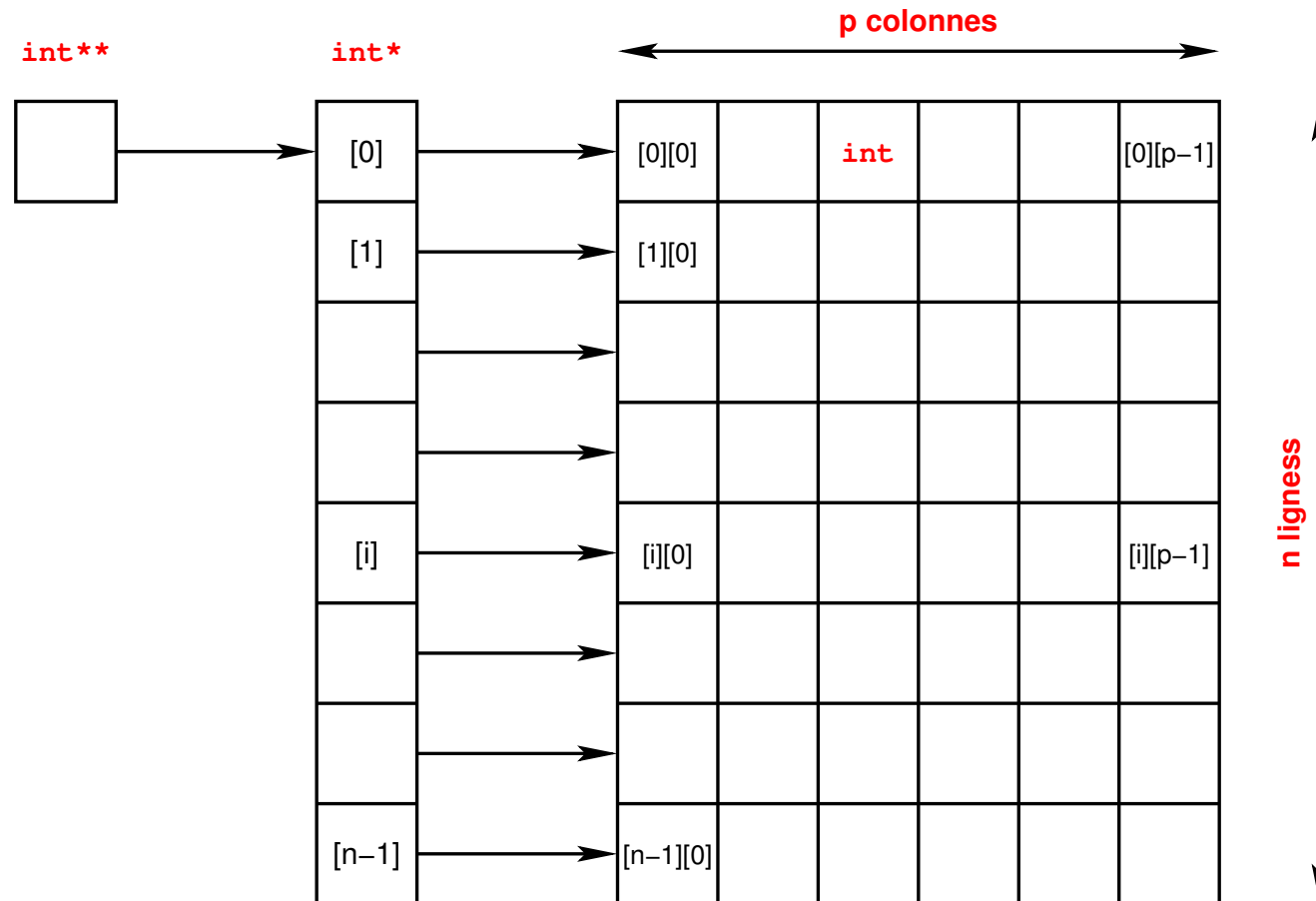
Allouer la mémoire sur le tas pour les $n \times p$ coefficients comme une **matrice aplatie en concaténant les n lignes** (appelée RowMajor dans lapack)



Puis, **structurer** cette zone mémoire linéaire en n blocs (lignes) de p cases (colonnes)
 ⇒ via n pointeurs de début de ligne ⇒ **allouer** aussi un vecteur de pointeurs
 ⇒ matrice accessible grâce à un **pointeur de pointeurs**

Allocation dynamique sur le tas d'un tableau 2D en C \Rightarrow **pointeur de pointeurs**

Tableau 2D en C = tableau de tableaux \Rightarrow pointeur de pointeurs




```
/* tableaux 2D de taille variable => point. de pointeurs */
/* car tableau 2D = tableau de tableaux */
/* tableau de pointeurs vers les débuts des lignes */
/* c'est à dire un pointeur de pointeurs */
/* lecture et impression d'une matrice de dim variables */
#include <stdio.h>
#include <stdlib.h>          /* fichier mat_point_point0.c */
/*****/
void print_int_mat(const int **tt, const int n, const int p) {
/* impression d'un tableau 2d d'entiers  n lignes x p col */
    int i, j ;
    printf(" tableau %d x %d avec point. de point.\n", n, p);
    for (i = 0 ; i < n; i++) {
        for (j = 0 ; j < p; j++) {
            printf("%d ", tt[i][j]);
        }
        printf("\n") ;
    }
}
/*****/
```

```
void double_int_mat(int **tt, const int n, const int p) {
/* doublement d'un tableau 2d d'entiers n lignes x p col */
    int i, j ;
    for (i = 0 ; i < n; i++) {
        for (j = 0 ; j < p; j++) {
            tt[i][j] *= 2;
        }
    }
}

/*****
int main(void) {
int nl, nc;
int ** plignes = NULL; /* tableau des pointeurs de début de ligne */
int * pmat = NULL ; /* pointeur sur les éléments de la matrice */
int iligne, col; /* indice de ligne , de colonne */
FILE* fpin = NULL;
fpin = fopen("matrice", "r"); /* fichier "matrice" à lire */
if (fpin == NULL) {
    fprintf(stderr, "erreur ouverture \n");
    exit(2);
}
}
*****/
```

```
/* lecture du fichier dans la matrice */
/* lecture des dimensions de la matrice */
fscanf(fpin, "%d %d", &nl, &nc);
/* (1) alloc globale (non fragmentée) de la matrice */
pmat = (int *) calloc((size_t) nc * nl , sizeof(int));
if ( pmat == NULL ) {
    fprintf(stderr, "erreur allocation globale\n");
    exit(3);
}
/* (2) alloc du tableau des pointeurs (int *) des débuts de ligne */
plignes = (int **) calloc((size_t) nl , sizeof (int*) );
if ( plignes == NULL ) {
    fprintf(stderr, "err allocation tableau de pointeurs\n");
    exit(3);
}
/* (3) initialisation du tableau des pointeurs de début de ligne */
for ( iligne=0; iligne < nl; iligne++ ) {
    plignes[ iligne ] = &(pmat[ iligne * nc ]);
}
/* NB: l'ordre (2) (1) (3) permet de se passer de pmat (=plignes[0]) */
```

```
/* lecture de la matrice */
for ( iligne=0 ; iligne<nl ; iligne++ ) {
    /* lecture d'une ligne du tableau */
    for ( col=0 ; col<nc ; col++ ) {
        /* lecture d'un entier du tableau */
        fscanf(fpin, "%d", &plignes[iligne][col]);
    }
}
fclose(fpin);
print_int_mat((const int **) plignes, nl, nc) ; /* impression du tableau
printf(" doublement des valeurs du tableau\n") ;
double_int_mat(plignes, nl, nc) ; /* doublement des valeurs */
print_int_mat((const int **) plignes, nl, nc) ; /* impression du tableau
/* attention à l'ordre des libérations pour éviter une fuite de mémoire
free (plignes[0]); /* libération du pointeur de la matrice */
free (plignes); /* libération du pointeur des pointeurs */
plignes = NULL; /* ne pointe plus vers une zone allouée */
exit(EXIT_SUCCESS) ;
} /* fichier mat_point_point0.c
```

9.5 La bibliothèque `libmnitab`

Allocation dynamique sur le tas en C : utiliser la bibliothèque `libmnitab`

⇒ directive préprocesseur `#include "mnitab.h"`

⇒ édition de liens avec l'option `-lmnitab`



Implémentations différentes sur les machines virtuelles 32 bits
et sur le serveur `sappli1` 64 bits

Cette bibliothèque contient de nombreuses fonctions permettant de gérer les tableaux de différents types en mémoire dynamique.

Exemples :

— Allocation et libération de tableaux 1D de doubles :

`double *double1d(int n)` pour allouer l'espace et

`void double1d_libere(double *vec)` pour libérer l'espace

— Allocation et libération de tableaux 2D de floats :

`float **float2d(int n, int p)` pour allouer l'espace et

`void float2d_libere(float **mat)` pour libérer l'espace

— d'autres fonctions de calcul de min, de max...

9.6 Allocation dynamique en fortran 2003

Changement du statut d'allocation d'un tableau allouable par une procédure :

- Impossible en fortran 95
sauf avec une option disponible sur tous les compilateurs
- Possible en **fortran 2003** : **argument muet tableau allouable**

Exemple : lecture d'une matrice dans un fichier via un sous-programme (lecture des dimensions, puis allocation, puis lecture des coefficients, puis retour de la matrice allouée et valorisée dans l'appelant, utilisation et libération).

Allocation dynamique au vol par affectation

En fortran 2003, un tableau allouable peut être alloué, voire réalloué **implicitement lors d'une affectation** (sous `gfortran v. ≥ 4.6`, pas sous `g95`).

Pas d'allocation au vol si le membre de gauche est une **section** de tableau (avec les séparateurs `:` d'indice).

`tab2 = tab1` \implies allocation ou réallocation de `tab2`

`tab2(:, :) = tab1` \implies pas d'allocation/réallocation de `tab2`

```
! argument tableau alloué par la procédure (f2003) ! proctalloc.f90
MODULE m_mat
IMPLICIT NONE
CONTAINS
SUBROUTINE lect_mat(matrice)
  ! argument tableau 2D alloué après lecture
  ! des dimensions de la matrice dans le fichier
  REAL, DIMENSION(:,:), ALLOCATABLE, INTENT(out) :: matrice
  INTEGER :: lignes, colonnes, i
  OPEN(file="mat.dat", unit=11, form="formatted")
  READ(11, *) lignes, colonnes
  ALLOCATE(matrice(lignes, colonnes))
  DO i = 1, lignes
    READ(11, *) matrice(i, :) ! lecture de la ligne i
  END DO
  CLOSE(11)
  RETURN
END SUBROUTINE lect_mat
END MODULE m_mat
```

```
PROGRAM matrices
USE m_mat
IMPLICIT NONE
REAL, DIMENSION(:, :), ALLOCATABLE :: mat
! tableau allouable, pas alloué ici
INTEGER :: i
! allocation et lecture de la matrice mat
CALL lect_mat(mat)
WRITE(*,*) "affichage de mat"
DO i=1, SIZE(mat, 1)
    WRITE(*,*) mat(i,:)
END DO
DEALLOCATE(mat)
! désallocation dans le programme ppal
END PROGRAM matrices
```


Allocation dynamique au vol par affectation (fortran 2003)

! allocation automatique par affectation ! t_alloc_affect2.f90

```
PROGRAM t_alloc_affect
```

```
INTEGER, DIMENSION(:), ALLOCATABLE :: v1, v2, v3
```

```
v1 = [1, 2] ! allocation de v1 par affectation
```

```
! => v1=[ 1 2 ]      taille = 2
```

```
v2 = [-3, -2, -1 ] ! allocation de v2 par affectation
```

```
! => v2=[ -3 -2 -1 ] taille = 3
```

```
v3 = v1 ! allocation implicite de v3 => 2 éléments
```

```
! => v3=[ 1 2 ]      taille = 2
```

```
v1 = v2 ! réallocation implicite de v1 => 3 éléments
```

```
! => v1=[ -3 -2 -1 ] taille = 3
```

```
v3(:) = v2 ! pas de réallocation de v3 => v2(3) inutilisé
```

```
! => v3=[ -3 -2 ]      taille = 2
```

```
DEALLOCATE(v1, v2, v3)
```

```
END PROGRAM t_alloc_affect
```

10 Chaînes de caractères

10.1 Introduction

Chaînes constantes dans les messages, les formats d'entrée/sortie...

Mais nécessité de **variables** pour effectuer des opérations sur les chaînes :

affectation, concaténation, classement, extraction de sous-chaînes, ...

par exemple pour générer des noms de fichiers, via des opérateurs ou des fonctions.

En fortran : **type chaîne de caractères paramétré** par sa longueur

délimiteurs de constante chaîne ' ou " : "aujourd'hui" ou 'aujourd'hui'

En C : **type caractère** seulement (délimiteur ')

⇒ chaîne = **tableau de caractères terminé par code nul**

et notation abrégée pour les chaînes constantes (délimiteur ") "hier"

Comme pour les tableaux, distinguer des chaînes de caractères de taille :

- **fixe** (à la compilation)
- **automatique** (portée limitée)
- **dynamique** (allocation sur le tas en C et fortran 2003)

10.2 Déclaration, affectation des chaînes de caractères

Langage C	Fortran 90
Pas de type de base	type intrinsèque paramétré
Tableau de caractères terminé par '\0' type const char *	CHARACTER (LEN=expres. constante)
Chaîne de longueur fixe	
char st1[4] = "oui" ; char st1[4]={'o','u','i','\0'}; un élément de plus pour le null	CHARACTER (LEN=3) :: st1="oui"
Longueur calculée à l'initialisation	
const char st2[] = "non";	CHARACTER (LEN=*) , PARAMETER :: st2="non"
Affectation globale	
st1[] = "non" ;/ interdite st1 = strcpy (st1, st2);	st1 = "non" st1 = st2

10.3 Manipulation des chaînes de caractères

Langage C	Fortran
Longueur	
sizeof(string) taille du tableau déclaré ($\geq n + 1$ avec <code>\0</code>) size_t strlen(const char *s) jusqu'au ' <code>\0</code> ' exclus ($\leq n$)	LEN(chaine) longueur telle que déclarée LEN_TRIM(chaine) longueur sans les espaces à droite
Concaténation	
char *strcat(char *dest, const char *src)	dest = dest // src // opérateur de concaténation
Comparaisons lexicographiques	
int strcmp(const char *s1, const char *s2) > 0 si s1 après s2 dans l'ordre lexicogr. = 0 si au même niveau dans l'ordre lexicogr.	LGE/LGT/LLE/LLT(chaine1, chaine2) LGE = Lexically Greater or Equal résultat booléen

Langage C	fortran 90
<pre>char *strchr(const char *s, int c);</pre> <p>1^{re} position du caractère c dans la chaîne s dernière position avec strchr</p>	<pre>SCAN(string, set [,back])</pre> <p>position de la 1^{re} occurrence d'un caractère de set dans la chaîne string position de la dernière si back=.true.</p>
<pre>char *strstr(const char *haystack, const char *needle);</pre> <p>1^{re} position de sous-chaîne dans chaîne</p>	<pre>INDEX(string, substring)</pre> <p>position de la 1^{re} occurrence de la sous-chaîne substring dans la chaîne string début de la dernière occurrence si back</p>
<pre>size_t strspn(const char *s, const char *accept);</pre> <p>longueur de la 1^{re} sous-chaîne de s constituée de caractères de accept</p>	<pre>SCAN(string, set [, back])</pre> <p>position de la 1^{re} occurrence d'un caractère de set dans la chaîne string position de la dernière si back=.true.</p>
<pre>size_t strcspn(const char *s, const char *reject);</pre> <p>longueur de la 1^{re} sous-chaîne de s constituée de caractères hors de reject</p>	<pre>VERIFY(string, set [, back])</pre> <p>position du premier caractère hors de l'ensemble set dans string dernier si back=.true.</p>

10.4 Chaînes de caractères en C

Prototypes des fonctions manipulant des chaînes dans `<string.h>` ⇒

`#include <string.h>`

Paramètres de **type** `char *` = pointeur de `char` permettant de manipuler des chaînes de longueur variable.

Préfixe `str` pour les fonctions

Pour éviter les accès à des zones mémoires arbitraires, préférer les **versions**

«**sécurisées**» avec argument entier **limitant le nombre de caractères** ⇒ `strn`

`strncpy`, `strncat`, ...

```
char *strncat(char *dest, const char *src, size_t n);
```

10.4.1 Déclaration de chaînes de longueur fixe en C

```
/* fichier char-elem_stat.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h> // contient le prototype de strlen, etc...
#define N 5
// ----- tableaux de caracteres de taille fixe -----
int main(void) {
    // declaration sans initialisation:
    char cfixe1[N]; // tableau de N-1 caracteres + '\0'
    // declaration avec initialisation:
    char cfixe2[N]={'0', 'u', 'i', '\0'}; // doit inclure '\0'
    // Declaration avec initialisation (methode conseillée):
    char cfixe3[N]= "Non"; // sans preciser le '\0'
    // Affichage de la taille des tableaux avec sizeof
    printf("Taille des tableaux via sizeof: %d\n\n", (int) sizeof(cfixe1));
}
```

```
// Affectation du premier tableau a l'aide d'une boucle:
printf("affichage caractere par caractere\n");
for(int i=0;i<N-1;i++){ // attention: indice de 0 a N-2
    cfixe1[i] = 'z';
    printf("%c", cfixe1[i]); // format %c pour un seul caractere
}
// Inclusion du caractere de fin de chaine ('\0') - impératif:
cfixe1[N-1] = '\0';
// Affichage de la taille (sans '\0') a l'aide de la fonction strlen:
printf("\n\nLongueur de la 1ere chaine:  %d\n", (int) strlen(cfixe1));
printf("Longueur de la 2eme chaine:    %d\n", (int) strlen(cfixe2));
printf("Longueur de la 3eme chaine:    %d\n", (int) strlen(cfixe3));
// Affichage des tableaux de caracteres (format %s pour une chaine):
printf("Affichage de la 1ere chaine:  %s\n", cfixe1);
printf("Affichage de la 2eme chaine:  %s\n", cfixe2);
printf("Affichage de la 3eme chaine:  %s\n", cfixe3);
exit(EXIT_SUCCESS);
}
```


Résultat à l'exécution :

```
Taille des tableaux via sizeof: 5
```

```
affichage caractere par caractere  
zzzz
```

```
Taille des tableaux via sizeof: 5
```

```
Longueur de la 1ere chaine: 4
```

```
Longueur de la 2eme chaine: 3
```

```
Longueur de la 3eme chaine: 3
```

```
Affichage de la 1ere chaine: zzzz
```

```
Affichage de la 2eme chaine: Oui
```

```
Affichage de la 3eme chaine: Non
```

10.4.2 Exemple de chaînes de caractères de taille variable en C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h> // contient prototype de strlen, etc...
int main(void) { // tableaux de caracteres de taille quelconque -----
    char *cvar1 = "Bonjour!"; // taille calculee a l'initialisation
    char cvar2[] = "Re-bonjour"; // taille calculee a l'initialisation
    char *cvar3 = NULL; // tableau dynamique de caracteres
    int nb; // nombre de caracteres
    printf("Nombre de caracteres (tableaux dynamique et automatique):\n");
    scanf("%d", &nb); // choix du nombre de caracteres par l'utilisateur
    cvar3 = (char *) calloc(nb+1, sizeof(char)); // allocation dynamique
    char cvar4[nb+1]; // declaration tardive d'un tableau automatique
    for(int i=0; i<nb; i++) { // attention: n indices de 0 a nb-1
        cvar3[i] = 'd'; // Affectation des tableaux
        cvar4[i] = 'a'; // a l'aide d'une boucle
    }
}
```

```
// Ajout du caractere de fin de chaine ('\0') - impératif
cvar3[nb] = '\0';
cvar4[nb] = '\0';
// Affichage de la taille (sans '\0') a l'aide de la fonction strlen:
printf("Longueur 1ere chaine: %d\n", (int) strlen(cvar1));
printf("Longueur 2eme chaine: %d\n", (int) strlen(cvar2));
printf("Longueur 3eme chaine (dynamique): %d\n", (int) strlen(cvar3));
printf("Longueur 4eme chaine (automatique): %d\n", (int) strlen(cvar4));
// Affichage des chaines de caracteres avec printf et %s :
printf("Affichage de la 1ere chaine: %s\n", cvar1);
printf("Affichage de la 2eme chaine: %s\n", cvar2);
printf("Affichage de la chaine dynamique : %s\n", cvar3);
printf("Affichage de la chaine automatique : %s\n", cvar4);
free(cvar3); // liberation de la memoire allouee dynamiquement
cvar3=NULL; // par precaution
exit(EXIT_SUCCESS);
}
```

Résultat à l'exécution :

```
Nombre de caracteres (tableaux dynamique et automatique) :  
5  
Longueur 1ere chaine: 8  
Longueur 2eme chaine: 10  
Longueur 3eme chaine (dynamique): 5  
Longueur 4eme chaine (automatique): 5  
Affichage de la 1ere chaine: Bonjour!  
Affichage de la 2eme chaine: Re-bonjour  
Affichage de la chaine dynamique : ddddd  
Affichage de la chaine automatique : aaaaa
```

10.4.3 Longueur d'une chaîne avec `strlen` et opérateur `sizeof`

Prototype : `size_t strlen(const char *s) ;`

Calcul de la longueur d'une chaîne **sans le caractère fin de chaîne.**

Ne pas confondre avec `sizeof` qui donne la taille du tableau :
au moins un élément de plus pour `'\0'`

Exemple :

```
char ch[]="oui";
```

dimensionné par le compilateur

```
char ch2[7]="non";
```

surdimensionné

```
printf("%d\n", sizeof(ch)); // => 4 ('o' 'u' 'i' '\0')
printf("%d\n", strlen(ch)); // => 3 ('o' 'u' 'i')
```

```
printf("%d\n", sizeof(ch2)); // => 7 ('n' 'o' 'n' '\0' +3)
```

```
printf("%d\n", strlen(ch2)); // => 3 ('n' 'o' 'n')
```

10.4.4 Concaténation de chaînes avec `strcat`

Prototype :

```
char *strcat (char *dest, const char *source) ;
```

Concatène (ajoute) la chaîne **source** à la chaîne **dest** et renvoie un pointeur sur **dest**. Gère le caractère `'\0'`.

Attention : **dest** doit être de longueur suffisante au risque de `segmentation fault`. La contrainte sur la taille de **dest** est :

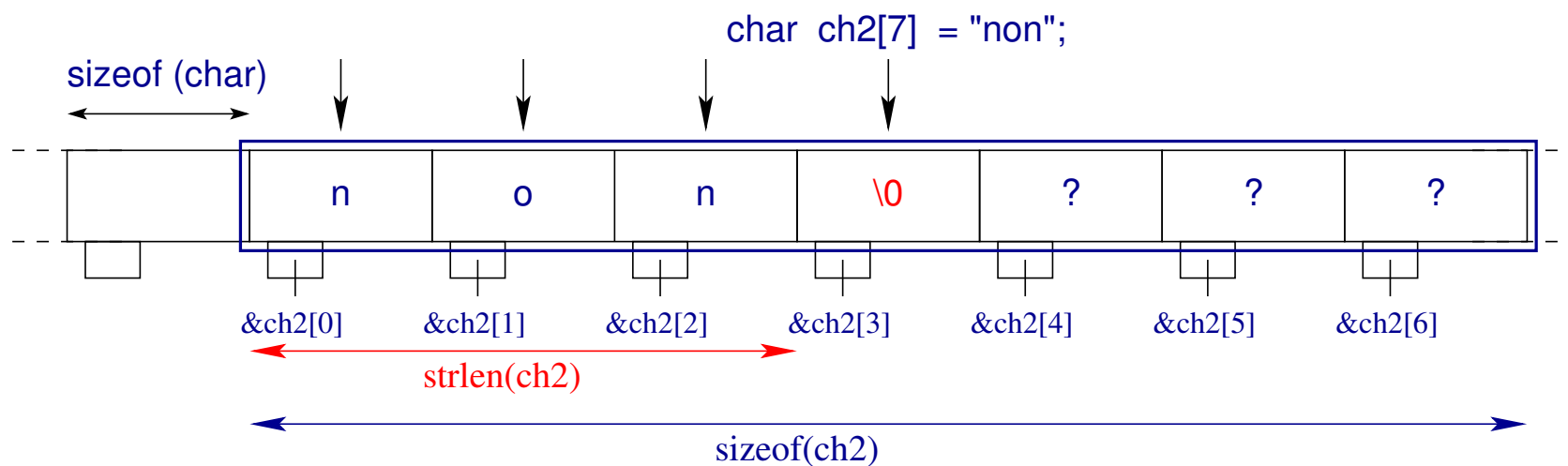
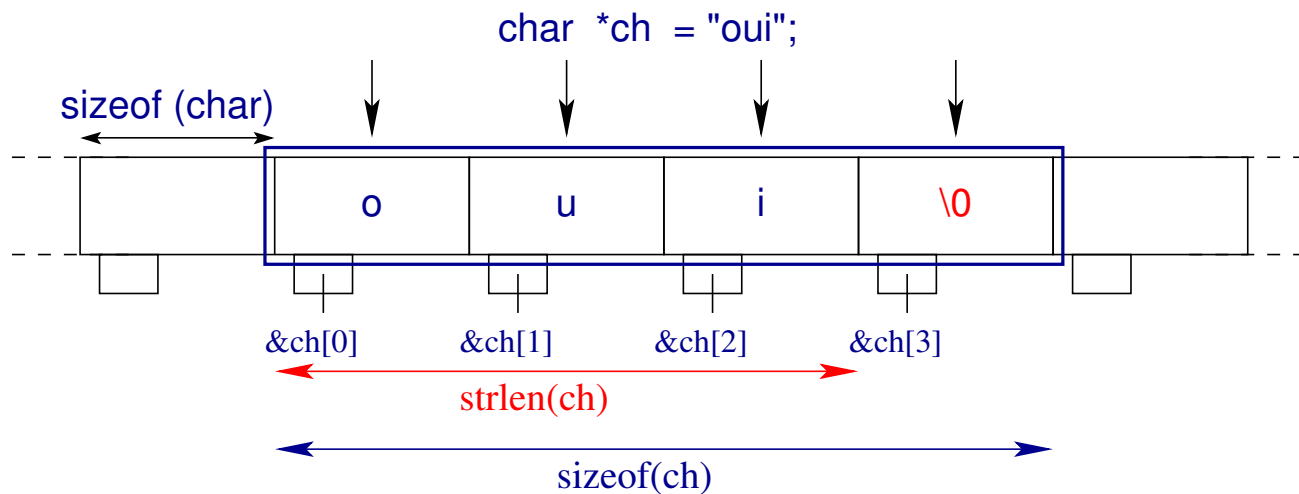
```
sizeof(dest) >= strlen(dest) + strlen(source) + 1
```

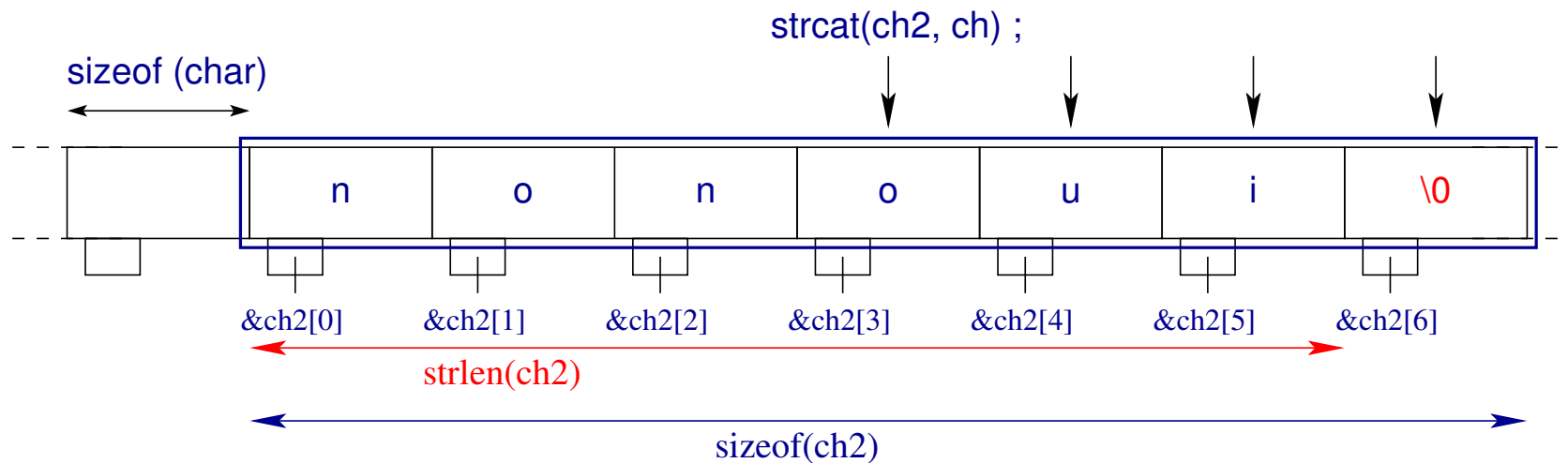
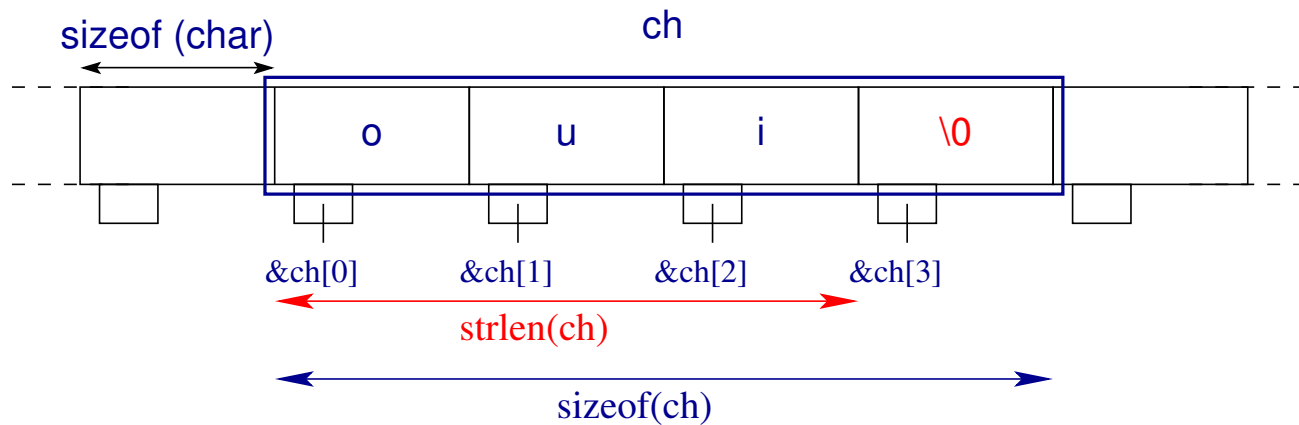
Exemple :

```
strcat (ch2, ch) ; // concatenation  
printf ("%s\n", ch2) ; // => nonoui  
printf ("%d\n", strlen(ch2)) ; // => 6
```

Préférer **strncat** avec 3^e argument = nb max de caractères copiés depuis `src` :

```
char *strncat (char *dest, const char *src, size_t n) ;
```





entrée		sortie
un caractère		
<pre>int getc(FILE *stream);</pre> 1 caractère	macro	<pre>int putc(int c, FILE *stream);</pre> 1 caractère
<pre>int getchar(void);</pre> = <code>getc(stdin)</code>	macro	<pre>int putchar(int c);</pre> = <code>putc(int c, stdout)</code>
<pre>int fgetc(FILE *stream);</pre>		<pre>int fputc(int c, FILE *stream);</pre>
une chaîne		
<pre>char *gets(char *s)</pre> ⚠ éviter (problème de sécurité)		<pre>int puts(const char *s)</pre>
<pre>char *fgets(char *s, int size, FILE *stream);</pre> au plus size caractères		<pre>int fputs(const char *s, FILE *stream);</pre>

10.4.5 Caractères non-ascii en C99 : caractères étendus et multi-octets

Rappels sur le codage des caractères :

- `ascii` sur 7 bits, `latin-1` sur 8 bits \Rightarrow sur 1 octet
- `unicode` : `utf-8` variable sur 1, 2, 3 ou 4 octets, `utf-32` sur 4 octets (fixe)

Deux représentations des caractères en C99 :

- type caractère étendu `wchar_t` (en pratique 4 octets comme UTF-32)
- caractère multi-octets = tableaux de 1, 2, 3, 4 `char` (cf. UTF-8) ou plus (6)

+ Fonctions de conversions `wcrtomb` (`wide-char` to `multi-byte`) et `mbrtowc`

`wcsrtombs` et `mbsrtowcs` (`s` pour les chaînes)

Variantes caractères étendus des fonctions sur 1 octet :

`wprintf` au lieu de `printf`

`wcsncpy` au lieu de `strncpy`

Format `%lc` pour les `wchar_t`

Préfixe `L` pour les constantes chaînes étendues `L"été"`

et le `wchar_t` de fin de chaîne `L'\0'` (sur 4 octets !)

10.5 Chaînes de caractères en fortran

10.5.1 Sous-chaînes en fortran

Notation similaire à celle des sous-sections de tableaux

chaîne (début : fin) (vide si *fin* < début)

character (len=7) :: chaîne="bonjour"

character (len=3) :: deb, fin

deb = chaîne(1:3) ⇔ deb = chaîne(:3)

fin = chaîne(5:7) ⇔ fin = chaîne(5:)

write(*,*) deb // "--" // fin affiche **bon--our**

10.5.2 Fonctions manipulant des chaînes en fortran

LEN : longueur telle que déclarée

LEN_TRIM : longueur sans blancs à droite

TRIM : suppression des **espaces terminaux**

TRIM(" ab cd ") ⇒ " ab cd"

ADJUSTL / **ADJUSTR** : **justification** à gauche/droite (même longueur)

ADJUSTL (" ab cd ") \Rightarrow "ab cd "

ADJUSTR (" ab cd ") \Rightarrow " ab cd"

TRIM (ADJUSTL (chaîne)) supprime les espaces des deux côtés

REPEAT : **duplication** de chaînes

REPEAT (" ab ", 3) \Rightarrow " ab ab ab "

INDEX **position** d'une **sous-chaîne** dans une chaîne

INDEX ("at**te**nte", "te") \Rightarrow **3** pour le 1^{er} **te** (**6** si **back=.true.**)

SCAN **position** dans une chaîne du premier **caractère** issu d'un ensemble

SCAN ("a**u**jour**u**rd' **h**ui", "ru") donne **2** pour le 1^{er} **u** (**10** si **back**)

VERIFY **position** dans une chaîne du premier **caractère hors** d'un ensemble

VERIFY ("au**j**our**u**rd' **h**ui", "aiou") donne **3** pour **j** (**9** pour **h** si **back**)

10.5.3 Tableaux de chaînes en fortran

Tableau de chaînes \Rightarrow rectangulaire \Rightarrow **même longueur**

```
CHARACTER (LEN=4) , DIMENSION (3) :: tab_ch
```

LEN(tab_ch) est le **scalaire commun** 4

LEN_TRIM(tab_ch) est un **tableau 1D** d'entiers (LEN_TRIM élémentaire)

tab_ch(1:2) section de tableau = deux premières chaînes de 4 caractères

tab_ch(2)(1:2) deux premiers caractères de la 2^e chaîne tab_ch(2)

10.5.4 Entrées-sorties de chaînes en fortran

Descripteur **A** \Rightarrow nombre de caractères = celui de l'expression chaîne

Descripteur **An** \Rightarrow nombre de caractères = n

(tronquer à droite ou compléter par des blancs à gauche)

10.5.5 Allocation dynamique de chaînes en fortran 2003

Standard **fortran 2003** seulement

Allocation **explicite** manuelle

```
CHARACTER (:), ALLOCATABLE :: ch1, ch2
```

```
ALLOCATE (CHARACTER(3) :: ch1) ! syntaxe particulière
```

```
ch1 = "123"
```

Allocation **implicite** au vol par affectation

```
ch2 = "texte" ! allocation au vol
```

```
ch2 = "si" ! réallocation au vol
```

```
deallocate(ch1, ch2)
```

10.5.6 Passage de chaînes en argument en fortran

Comme pour les tableaux, **ne pas passer la longueur** et utiliser **LEN** dans la procédure

⇒ argument formel `character (len=*)` dans la procédure.

11 Entrées-sorties

11.1 Type de fichiers et accès : avantages respectifs

fichiers	formatés	binaires
	saisie et affichage	comme en mémoire
compacité	—	+
rapidité des E/S	—	+
précision conservée	—	+
portabilité	+	—

accès aux données	séquentiel	direct
	lire/écrire dans l'ordre	enregistrements indexés

Ouverture d'un flot en C		connexion d'un fichier à une unité logique en f90	
<pre>FILE *fopen(const char *path, const char *mode)</pre>		<pre>OPEN(UNIT=unit, FILE=filename & [, STATUS=mode] [, IOSTAT=i] ...) FORM="unformatted" si binaire FORM="formatted" si texte (défaut) ACCESS="sequential"(défaut)/"direct"</pre>	
<pre>mode</pre>	<pre>position</pre>	<pre>ajouter + si mise à jour, b si binaire</pre>	<pre>"old" mode = "new" "replace"</pre>
rend NULL si erreur		IOSTAT ≠ 0 si erreur	
fermeture (et vidage du tampon)			
<pre>int fclose(FILE *stream)</pre>		<pre>CLOSE(unit [, IOSTAT=ivar, ...])</pre>	

11.2 Entrées-sorties binaires (ou non formatées)

Même codage que dans la mémoire vive de la machine

C	fortran 90
entrée	
<pre>size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)</pre>	<pre>READ(unit, [, iostat]...) & liste de variables</pre> <p>attend balises en fin/début d'enregistrement avec le nombre d'octets ⚠</p>
sortie	
<pre>size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)</pre>	<pre>WRITE(unit, [, iostat]...) & liste d'expressions</pre> <p>ajout de balises en fin/début d'enregistrement avec le nombre d'octets ⚠</p>

11.3 Entrées-sorties formatées (fichiers codant du texte)

3 fonctions en C, 1 fonction en fortran

écriture	C	fortran 90
stdout	<code>printf(...)</code> = <code>fprintf(stdout,)</code>	<code>write(*, ...)</code> ou <code>print ...</code> ,
fichier	<code>fprintf(FILE* stream, ...)</code>	<code>write(unit, ...)</code>
chaîne	<code>sprintf(char* string, ...)</code>	<code>write(chaine, ...)</code>
lecture	C	fortran 90
stdin	<code>scanf(...)</code> = <code>fscanf(stdin,)</code>	<code>read(*, ...)</code> ou <code>read ...</code> ,
fichier	<code>fscanf(FILE* stream, ...)</code>	<code>read(unit, ...)</code>
chaîne	<code>sscanf(char* string, ...)</code>	<code>read(chaine, ...)</code>

les opérations de **lecture et de conversion s'effectuent conjointement**

=> impossible de récupérer une erreur de conversion.

Pour fiabiliser les saisies interactives, il faut **décomposer le processus** en

1. lisant dans une chaîne de caractères
2. décodant ensuite la chaîne

C	fortran 90
entrée	
<pre>int fscanf(FILE *stream, const char *format, liste de pointeurs)</pre> <p>valeur de retour = nb de conversions</p>	<pre>READ(unit, format & [, IOSTAT=ok...]) & liste de variables</pre> <p>IOSTAT = statut d'erreur (0 si correct)</p>
sortie	
<pre>int fprintf(FILE *stream, const char *format, liste d'expressions)</pre> <p>valeur de retour = nb de caractères</p>	<pre>WRITE(unit, format & [, IOSTAT=ok...]) & liste d'expressions</pre> <p>IOSTAT = statut d'erreur (0 si correct)</p>

Séparation en lignes

c	fortran 90
<p>pas de changement d'enregistrement lors d'une instruction <code>printf</code> spécifier <code>\n</code> en sortie pour changer de ligne</p>	<p>forcer par <code>/</code> dans le format changement d'enregistrement à chaque ordre READ ou WRITE sauf si <code>advance='no'</code></p>

11.4 Formats d'entrée–sortie

Correspondance très approximative entre C et fortran (w =largeur, p = précision)

	c	fortran 90
entiers		
décimal	%w[.p]d	I_w [.p]
	%d	I0
binaire		B_w
octal	%wo	O_w
hexadécimal	%wx	Z_w
réels		
virgule fixe	%w[.p]f	F_{w . p}
virgule flottante float/double (printf) float (scanf)	%w[.p]e	E_{w . p} ES_{w . p} (scientif.) EN_{w . p} (ingénieur)
général	%w[.p]g	G_{w . p}
caractères		
caractère	%w[.p]c	A [w]
chaîne	%w[.p]s	A [w]

C	fortran
format libre	
non (sauf C++ : <code>cin/cout</code>)	oui : *
largeur w	
facultative (%d, %f, %e, %g, %s)	obligatoire sauf A, I0, F0, G0 en F2008
si largeur w insuffisante	
⇒ élargie pour écrire	conservée ⇒ écrit ***
si nombre de descripteurs \neq nb d'éléments de la liste d'E/S	
le nb de descripteurs prime ⇒ risque d'accès mémoire non réservée	le nb d' éléments de la liste prime ⇒ relecture ou abandon du format

11.5 Exemple de lecture de fichier formaté en C

```
#include <stdio.h> /* lecture.c */
#include <stdlib.h>
/* lecture du fichier donnees */
int main(void) {
char nom[80] ; /* limitation des chaînes à 80 caractères */
char article[80] ;
int nombre ;
float prix, dette ;
int n , ligne=1;
FILE *fp = NULL ; /* pointeur sur le flot d'entrée */
char fichier[] ="donnees"; /* nom du fichier formaté */
if ( (fp = fopen(fichier,"r")) == NULL ) { /* ouverture du fichier */
    fprintf(stderr, "erreur ouverture du fichier %s\n", fichier) ;
    exit (EXIT_FAILURE) ;
}
printf("ouverture correcte du fichier %s\n", fichier) ;
```

```
while( (n=fscanf(fp, "%s %s %d %f", nom, article, &nombre, &prix) ) != EOF)
    /* boucle de lecture des données jusqu'à EOF = End Of File */
    { if (n == 4) { /* si fscanf a réussi à convertir 4 variables */
        dette = nombre * prix ;
        printf("%s %s \t %d x %6.2f = %8.2f\n", nom, article, nombre, prix, dette);
        ligne++;
    }
    else {
        fprintf(stderr, "problème fscanf ligne %d\n", ligne);
        exit (EXIT_FAILURE) ;
    }
}
printf("fin de fichier %d lignes lues \n", ligne-1); /* sortie normale p
fclose(fp) ; /* fermeture du fichier */
exit(EXIT_SUCCESS) ;
} /* lecture.c */
```


donnees

```
dupond cafe 5 10.5
durand livre 3 60.2
jean disque 5 100.5
paul cafe 6 12.5
jean disque 4 110.75
julie livre 9 110.5
```

resultat

```
ouverture correcte du fichier donnees
dupond cafe      5 x 10.50 = 52.50
durand livre     3 x 60.20 = 180.60
jean disque     5 x 100.50 = 502.50
paul cafe       6 x 12.50 = 75.00
jean disque     4 x 110.75 = 443.00
julie livre     9 x 110.50 = 994.50
fin de fichier 6 lignes lues
```

11.6 Exemple d'écriture de tableau 1D en fortran

```

PROGRAM format_tab
IMPLICIT NONE
INTEGER, DIMENSION(3) :: t = (/ 1, 2, 3 /)      ! tableau initialisé
INTEGER :: i
WRITE(*, '(3i4)') t(:)                          ! tableau global => en ligne
DO i = 1, 3                                     ! boucle explicite
    WRITE(*, '(i4)') -t(i)                      ! => un chgt d'enregistrement par ord
END DO                                          ! => affichage en colonne
WRITE(*, '(3i4)') (2*t(i), i=1, 3)             ! boucle implicite => affichage en li
WRITE(*, '(i4)') (-2*t(i), i=1, 3)             ! format réexploré => en colonne
DO i = 1, 3                                     ! boucle explicite
    WRITE(*, '(i4)', advance='no') 3*t(i)      ! mais option advance='no'
END DO                                          ! => affichage en ligne
WRITE(*, *)                                     ! passage à l'enregistrement suivant
END PROGRAM format_tab

```

Affichage d'un tableau 1D		
t	en ligne avec le tableau global	1 2 3
-t	en colonne avec la boucle explicite	-1 -2 -3
2*t	en ligne avec la boucle implicite	2 4 6
-2*t	en colonne malgré la boucle implicite, car le format ne satisfait qu'un élément de la liste ⇒ format réexploré ⇒ changement de ligne	-2 -4 -6
3*t	en ligne malgré la boucle explicite, grâce à ADVANCE='no' .	3 6 9

12 Structures ou types dérivés

12.1 Intérêt des structures

Tableau = agrégat d'objets de **même type** repérés par un ou des **indices entiers**

Structure/type dérivé = agrégat d'objets de **types différents** (chaînes de caractères, entiers, flottants, tableaux...) repérés par un **nom de champ/composante**

⇒ représentation de données composites et manipulation champ par champ ou globale (passage en argument des procédures simplifié par **encapsulation**)

Représentation à l'image des données dans certains fichiers (tableau de structures)

Exemples

- étudiant = nom, prénom, numéro, date de naissance, UE et notes associées...
- échantillon de mesure sous ballon sonde à un instant donné=
altitude, pression, température, humidité, vitesse, orientation du vent
- point d'une courbe = abscisse, ordonnée, lettre, couleur

Imbrication possible → **structures de structures** :

personne = nom, prénom, date de naissance

étudiant = personne, numéro, tableau d'UE et de notes

Structures statiques : champs/composantes de **taille fixe**

Structures dynamiques en fortran : comportant des champs de **taille variable**

(tableaux ou chaînes de caractères allouables par exemple) :

Exemple : la liste des couples nom d'UE + note dépend de l'étudiant

Structures auto-référencées parmi les champs d'une structure, introduire des **pointeurs vers un objet du même type** :

→ objets dynamiques complexes tels que arbres, **listes chaînées**

⇒ modéliser des données dont le nombre évolue pendant leur manipulation

1. lors de la saisie des données
2. pour insérer ou supprimer dans une liste ordonnée lors d'un classement

tableaux dynamiques insuffisants ⇒ **allocation élément par élément**

+ pointeurs entre voisins

12.2 Définition, déclaration et initialisation des structures

12.2.1 Définition de structures/types dérivés

Langage C : structure	Fortran 90 : type dérivé
Définition d'un type point	
pas de déclaration d'objet \Rightarrow ne réserve pas d'espace	
<pre> struct point { int no ; // 1^{er} champ float x ; // 2^e champ float y ; // 3^e champ }; </pre>	<pre> TYPE point integer :: no ! 1^{re} composante real :: x ! 2^e composante real :: y ! 3^e composante END TYPE point </pre>
<pre> typedef struct point spoint; spoint synonyme de struct point </pre>	
Où les déclarer ?	
<p>dans un fichier d'entête puis #include</p>	<p>dans un module (avant CONTAINS) puis USE</p>

12.2.2 Déclaration et initialisation d'objets de type structure

Langage C	Fortran 90
Structure	Type dérivé
Déclaration de variables de type structure point	
<code>struct point debut, fin ;</code> ou <code>spoint debut, fin;</code>	<code>TYPE (point) :: debut, fin</code>
Initialisation via un constructeur	
<code>struct point fin={9,5.,2.};</code>	<code>TYPE (point) :: fin=point (9,5.,2.)</code>
Implémentation des structures	
Contraintes d'alignement en mémoire ⇒ compléter parfois avec des octets de remplissage	
La taille de la structure n'est pas toujours la somme des tailles	
utiliser <code>sizeof</code> si allocation	
ordre des champs respecté	ordre des compos. modifiable sauf si attribut SEQUENCE

12.2.3 Contraintes d'alignement dans les structures en C

La taille de la structure **n'est pas toujours** la somme des tailles des champs

Exemple de structure impliquant

3 octets de remplissage (*padding*)

signalés avec option **-Wpadded**

```
typedef struct {
    int n ; // 4 octets
    char c; // 1 octet
} lettre; // 5+3=8 octets
```

⇒ utiliser **sizeof** de la structure **si allocation**

```
p=(struct lettre *) calloc(n_struct, sizeof(struct lettre));
```

Options de compilation gcc pour spécifier le remplissage **-fpack-struct=**

gcc sans option ⇒ **sizeof(struct lettre)** → 8

gcc -fpack-struct=**2** ⇒ **sizeof(struct lettre)** → 6

gcc -fpack-struct=**1** ⇒ **sizeof(struct lettre)** → 5

Mais éviter car code sous-optimal et incompatible avec code sans cette option.

Éviter ou limiter le padding en classant les petits champs à la fin
(en particulier pour de l'interopérabilité C/fortran)

structure mal ordonnée (12 octets)

```
struct lettre1 {
    char c1; // 1 + 3
    int n ; // 4
    char c2; // 1
    char c3; // 1
    char c4; // 1 + 1
} ; // 12
```

structure bien ordonnée (8 octets)

```
struct lettre2 {
    int n ; // 4
    char c1; // 1
    char c2; // 1
    char c3; // 1
    char c4; // 1
} ; // 8
```

12.2.4 Implémentation des types dérivés en fortran

L'ordre des composantes peut être modifié par l'implémentation

sauf avec l'attribut **SEQUENCE** (interdit si `bind(c)` en interopérabilité)

-fpack-derived pour compacter les types dérivés avec g95/gfortran

12.3 Manipulation des structures

Langage C	Fortran 90
Accès aux champs d'une structure / composantes d'un type dérivé	
<pre>debut.no = 1 ; debut.x = 0. ; debut.y = 0. ;</pre>	<pre>debut%no = 1 debut%x = 0. debut%y = 0.</pre>
Affectation globale (même type)	
<pre>fin = debut ; constructeur interdit (sauf initialis.)</pre>	<pre>fin = debut constructeur fin=point (9, 5., 2.)</pre>
Opérations sur les structures	
Opérateurs natifs ==, + ... non applicables	
définir des fonctions	définir des fonctions et surcharger les opérateurs natifs
entrées/sorties formatées	
champ par champ	composante par composante ou globales : format libre (*) ou DT personnalisé (* sauf si composantes allouables)

```

MODULE m_point ! module de définition du type point ! type_pt.f90
  IMPLICIT none
  TYPE point
    INTEGER :: no ! numéro
    REAL    :: x ! abscisse
    REAL    :: y ! ordonnée
  END TYPE point
END MODULE m_point

PROGRAM points ! programme principal
  USE m_point ! visibilité du type dérivé
  TYPE(point) :: a, b ! déclaration de deux "points"
  a = point(5, .4, -2.) ! construction de a
  WRITE(* , *) "a = ", a ! affichage global de a
  b = a ! affectation globale de b
  b%no = 6 ! modification d'une composante
  WRITE(* , *) "b = ", b%no, b%x, b%y ! affichage par composantes
END PROGRAM points ! type_pt.f90

```

```
#include <stdio.h>                                     /* fichier type_pt.c */
#include <stdlib.h>
struct point { /* définition globale de la structure point */
    int no; /* numéro */
    float x; /* abscisse */
    float y; /* ordonnee */
};
typedef struct point spoint; /* type spoint = raccourci */
int main(void) {
    spoint a = {5, .4 , -2.}; /* définition avec constructeur de a */
    spoint b; /* déclaration de type spoint */
    printf("taille de la struct spoint %d\n", (int) sizeof(spoint));
    printf("a = %d %g %g\n", a.no, a.x, a.y); /* aff. des champs de a */
    b = a; /* affectation globale */
    b.no = 6; /* modification d'une composante */
    printf("b = %d %g %g\n", b.no, b.x, b.y); /* aff. des champs de b */
    exit(EXIT_SUCCESS);
} /* fichier type_pt.c */
```

12.4 Structures et tableaux

Langage C	Fortran 90
Structures contenant des tableaux de taille fixe	
<pre>struct point3d { char nom[5]; float coord[3]; } ; struct point3d p1;</pre>	<pre>TYPE point3d CHARACTER(len=4) :: nom REAL, DIMENSION(3) :: coord END TYPE point3d TYPE(point) :: p1</pre> <p style="text-align: center;">ordonnée de p1</p> <pre>p1.coord[1]</pre>
Tableaux de structures	
<pre>struct point courbe[9] ;</pre> <p style="text-align: center;">abscisse du premier élément du tableau</p> <pre>courbe[0].x = 2. ;</pre>	<pre>TYPE(point), dimension(9) :: courbe</pre> <pre>courbe(1)%x = 2.</pre>

12.5 Structures/types dérivés, pointeurs et procédures

Langage C	Fortran 90
Pointeur vers une structure / vers un type dérivé	
<code>struct point *pstr ;</code>	<code>TYPE (point) , POINTER :: pstr</code>
Raccourci <code>-></code>	<code>pstr%x</code> désigne la composante
<code>pstr->x</code> \iff <code>(*pstr).x</code>	rappel : le pointeur désigne sa cible
valeur de retour d'une fonction	
<code>spoint psym(struct point m) ;</code>	<code>type (point) FUNCTION psym(m)</code> <code>type (point) , intent (in) :: m</code>
arguments passés	
par copie de valeur	par référence
\Rightarrow pointeur vers la structure si	SUBROUTINE <code>sym(m, n)</code>
la fonction doit modifier un des champs	<code>type (point) , intent (in) :: m</code>
<code>void sym(spoint m, spoint *pn) ;</code>	<code>type (point) , intent (out) :: n</code>

```
MODULE m_point ! module de définition du type point ! fichier sym2_pt.  
  IMPLICIT NONE  
  TYPE point  
    INTEGER :: no ! numéro  
    REAL    :: x ! abscisse  
    REAL    :: y ! ordonnée  
  END TYPE point  
END MODULE m_point
```

```
MODULE m_sym ! module des procédures de calcul du symétrique/diagonale  
  USE m_point ! pour importer le type point dans le module  
  CONTAINS  
  FUNCTION psym(m) ! fonction de calcul du symétrique  
    TYPE(point) :: psym ! résultat de type point  
    TYPE(point), INTENT(in) :: m  
    psym = point (-m%no, m%y, m%x) ! chgt signe de no et échange x/y  
  END FUNCTION psym
```

```

SUBROUTINE sym(m, n)  ! sous programme de calcul du symétrique
    TYPE(point), INTENT(in) :: m
    TYPE(point), INTENT(out) :: n
    n = point (-m%no, m%y, m%x) ! chgt de signe du no et échange x/y
END SUBROUTINE sym
END MODULE m_sym
PROGRAM sym_point      ! programme principal
    ! USE m_point  ! pas nécessaire car USE m_sym implique USE m_point
    USE m_sym  ! pour connaître l'interface des procédures sym et psym
    IMPLICIT NONE
    TYPE(point) :: a, b      ! déclaration de deux "points"
    a = point(5, 1., -2.) ! construction de a
    WRITE(* , *) "a = ", a      ! affichage global de a
    WRITE(* , *) "psym(a) = ", psym(a) ! avec la fonction => -5 -2. 1.
    CALL sym(a, b)           ! appel de sym => -5 -2. 1.
    WRITE(* , *) "sym. de a = ", b ! affichage global de b
END PROGRAM sym_point      ! fichier sym2_pt.f90

```



```
#include <stdio.h>                                /* fichier sym2_pt.c */
#include <stdlib.h>
struct point { /* définition globale de la structure point */
    int no; /* numéro */
    float x; /* abscisse */
    float y; /* ordonnee */
};
typedef struct point spoint; /* type spoint = raccourci */
spoint psym(spoint m);
spoint psym(spoint m) { /* fonction à valeur de type spoint */
    spoint symetrique;
    symetrique.no = -m.no; /* changement de signe */
    symetrique.x = m.y; /* échange entre x et y */
    symetrique.y = m.x;
    return symetrique;
}
```

```
void sym(spont m, spont *n); /* passer le pointeur pour pouvoir */
void sym(spont m, spont *n){ /* modifier les champs de la structure */
    n->no = -m.no; /* chgt de signe de no et échange x/y */
    n->x = m.y;
    n->y = m.x;
}
int main(void) {
    spont a = {5, 1. , -2.}; /* définition de a */
    spont b; /* déclaration de type spont */
    printf("taille de la struct spont %d\n", (int) sizeof(spont));
    printf("a = %d %g %g\n", a.no, a.x, a.y); /* affich. des champs de a */
    printf("psym(a) = %d %g %g\n", psym(a).no, psym(a).x, psym(a).y);
    sym(a, &b); /* appel de sym */
    printf("sym. de a = %d %g %g\n", b.no, b.x, b.y); /* affichage de b */
    exit(EXIT_SUCCESS);
} /* fichier sym2_pt.c */
```

12.6 Exemple de structures auto-référencées : listes chaînées

Listes simplement chaînées (*singly linked lists*)

Collection ordonnée et de taille arbitraire d'éléments de même type :
chaque élément (nœud) de la liste chaînée est une structure qui comporte un
pointeur vers l'élément suivant.

Définition de la structure	
Langage C	Fortran 90
<pre>struct point{ float x; float y; struct point *next; };</pre>	<pre>type point real :: x real :: y type(point), pointer :: next end type point</pre>

La structure comporte un **pointeur** vers un objet du type qu'elle définit

⇒ structure **auto-référencée**

Construction de proche en proche de la liste simplement chaînée, par exemple à la lecture d'un fichier, dans l'ordre de lecture :

Initialisation : allocation du premier élément

Corps de la boucle

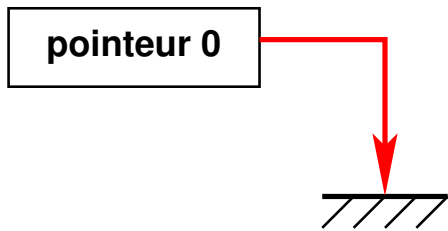
1. affectation des données de la structure courante (sauf le pointeur)
2. allocation de l'élément suivant
3. affectation du pointeur du courant vers le suivant (chaînage simple)

En fin de boucle : le dernier élément doit pointer vers **NULL**.

Le premier élément permet de retrouver toute la chaîne.

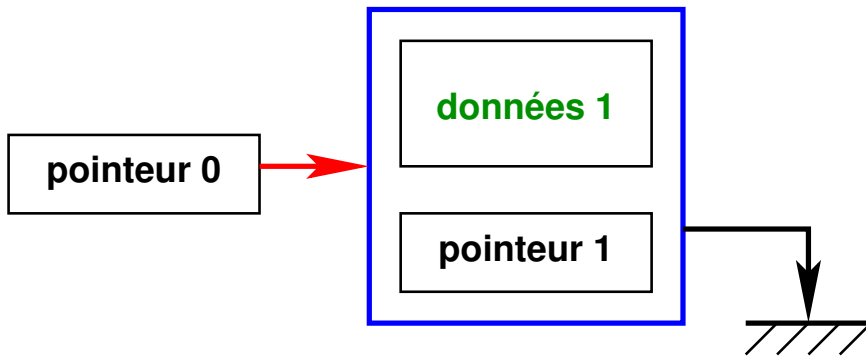
Associer des **procédures récursives** pour parcourir la chaîne et agir sur la liste

Structure adaptée à l'insertion et la suppression d'éléments (**classement** par ex.) mais ne pas perdre le chaînage !

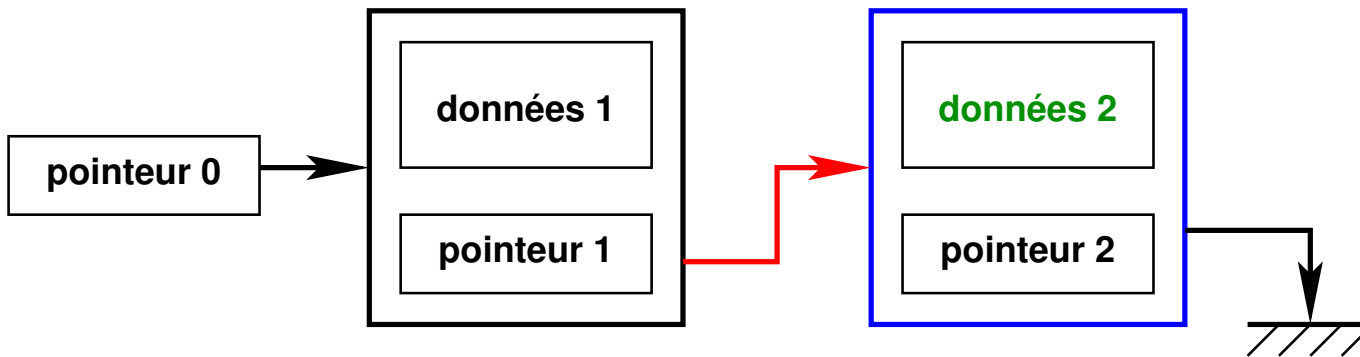


Liste vide :
pointeur de tête nul

Construction d'une liste simplement chaînée : deux premiers éléments

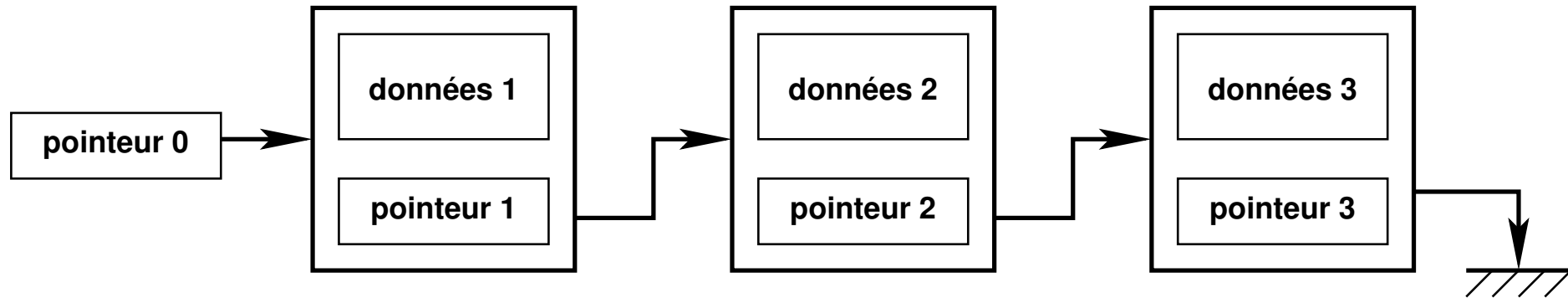


Allocation élément 1
Tête pointe vers él. 1
Affectation des données 1
Pointeur 1 nul

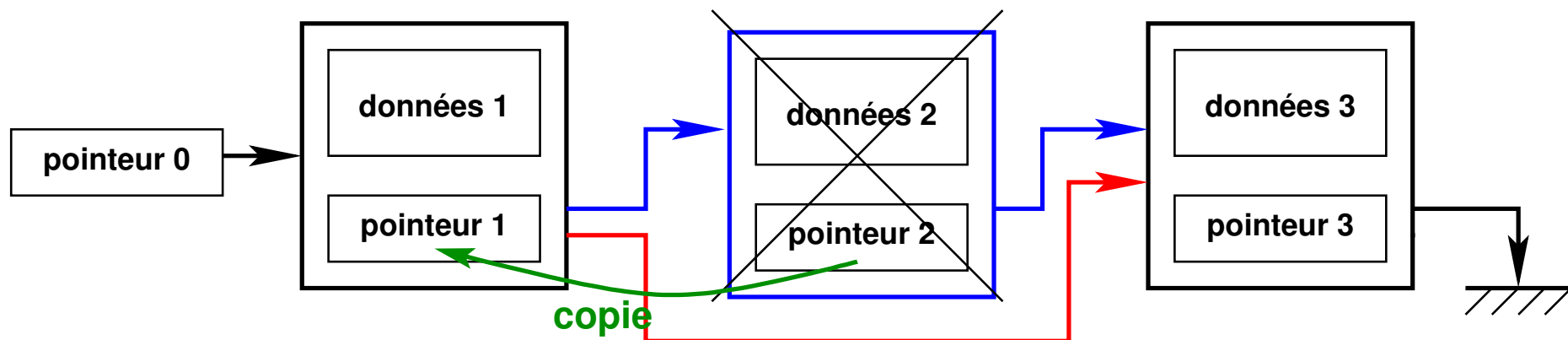


Allocation élément 2
Pointeur 1 vers él. 2
Affectation des données 2
Pointeur 2 nul

Liste chaînée à 3 éléments



Suppression d'un élément intérieur (en bleu) dans une liste chaînée



faire pointer (1) vers nœud (3), puis désallouer (2)

Listes doublement chaînées (*doubly linked lists*)

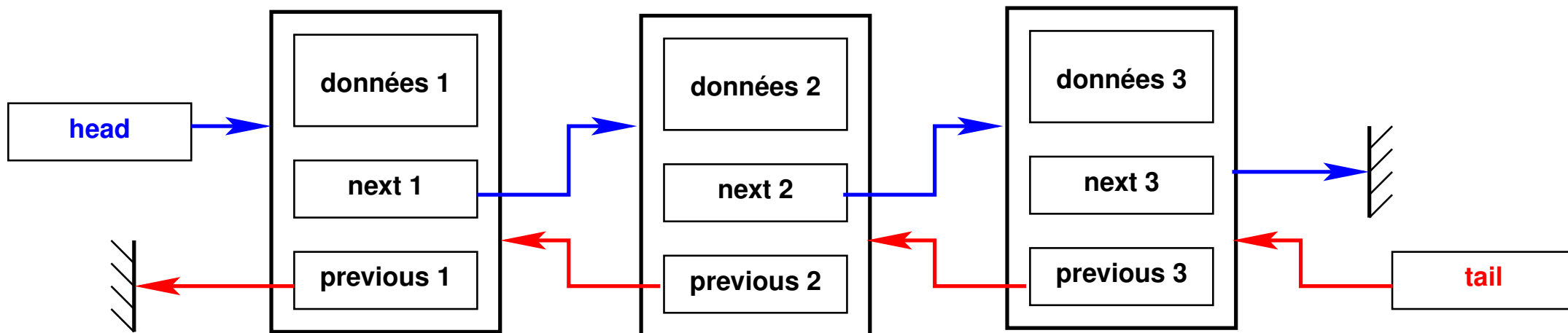
Problème de la liste simplement chaînée : les noeuds de fin de liste ne sont accessibles qu'en parcourant tout le début.

⇒ compléter par un chaînage en sens inverse

Deux pointeurs dans la structure : sur le suivant (*next*) et sur le précédent (*previous*).


Liste accessible depuis les deux extrémités (*head* et *tail*).

Liste doublement chaînée à 3 éléments



12.7 Structures à composantes dynamiques

Langage C	Fortran 2003
Définition de la structure avec des champs dynamiques	
<p>pointeur vers le tableau</p> <pre>struct point_ndim { int n; numéro int dim; nb de dim. float* coord; vecteur } ; struct point_ndim a, b;</pre>	<p>tableau allouable</p> <pre>TYPE point_ndim INTEGER :: n numéro REAL, DIMENSION(:), & ALLOCATABLE :: coord END TYPE point_ndim TYPE(point_ndim) :: a, b</pre>
Allocation	
<p>à prendre en charge par l'utilisateur</p> <pre>a.n = 5; a.dim = 2; a.coord = calloc(2, sizeof float); a.coord[0] = 1.; a.coord[1] = -3.;</pre>	<pre>a%n = 5 allocate (a%coord(2)) a%coord(1) = 1. a%coord(2) = -3. automatique par le constructeur a = point_ndim(5, [1., -3.])</pre>
Libération	
à prendre en charge sinon fuite de mémoire	automatique en sortie de portée

Langage C	Fortran 2003
Affectation globale	
<p>copie superficielle</p> <pre>struct point_ndim a, b; b = a;</pre> <p> ne recopie que les champs (donc le pointeur) mais pas les variables pointées ⇒ à prendre en charge par une fonction de recopie avec allocation</p>	<p>copie profonde</p> <pre>TYPE (point_ndim) :: a, b b = a</pre> <p>recopie les composantes donc les valeurs en réallouant à la volée si nécessaire</p> <pre>b = point_ndim(7, [-1., 3., 2.]) b = a réalloue à la nouvelle taille</pre>
<p>NB : pas de tableaux automatiques dans les structures en C</p>	

12.8 Conclusion sur les structures

Les structures permettent de regrouper des données hétérogènes pour les communiquer de façon plus concise entre procédures.

La notion de structure devient beaucoup plus puissante pour manipuler des objets complexes si on lui associe des **méthodes de manipulation** sous forme de

- **fonctions** : possible dans les deux langages
- **opérateurs** : possible en fortran, pas en C, mais en C++
en fortran, surcharge d'opérateurs y compris affectation

Association structures de données et méthodes \Rightarrow programmation objet

13 Éléments de compilation séparée

Découper le code en plusieurs fichiers sources

(une ou quelques procédures par fichier)

⇒ séparer

(1) phase des compilations et

(2) phase de l'édition de liens

Rappel

unité de compilation	
Fortran	langage C
le module	le fichier

13.1 Intérêt de la compilation séparée

- modularisation du code
- mise au point plus rapide (ne recompiler que partiellement)
- réutilisation des procédures
- création de bibliothèques (collections de fichiers objets)
- automatisation avec l'utilitaire **make**

13.2 Mise en œuvre robuste de la compilation séparée

Donner les moyens au **compilateur** de vérifier si **le nombre, le type et la position des arguments des procédures** lors d'un appel sont conformes à l'interface ou prototype de la procédure.

Fortran	langage C
passage par référence donc respect exact du type, mais généricité	passage par copie donc conversion des arguments sauf pour les pointeurs

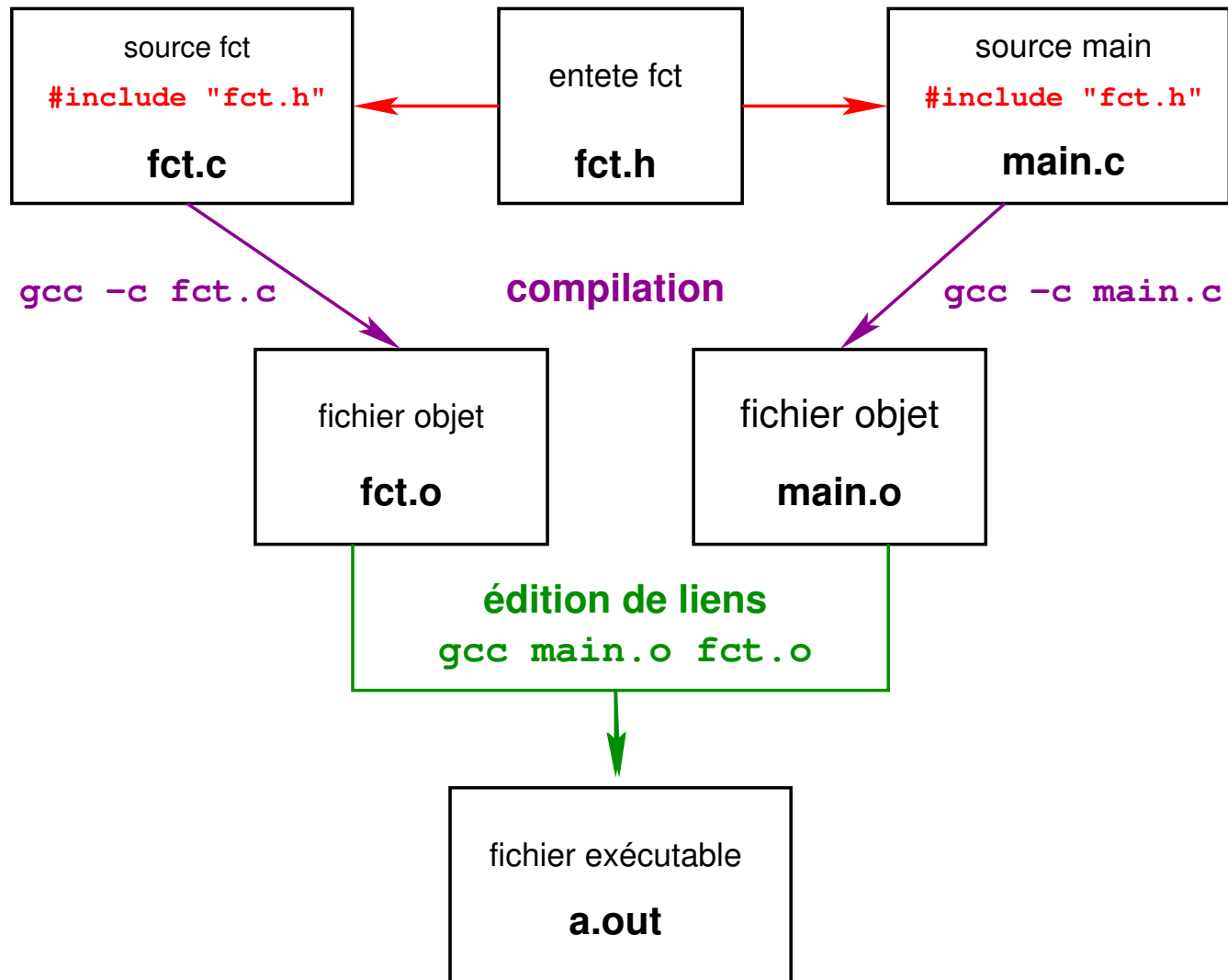
Première solution : dupliquer l'information sur l'interface en la déclarant au niveau des procédures qui l'utilisent.

Fortran	langage C
déclaration d'interface	déclaration de prototype

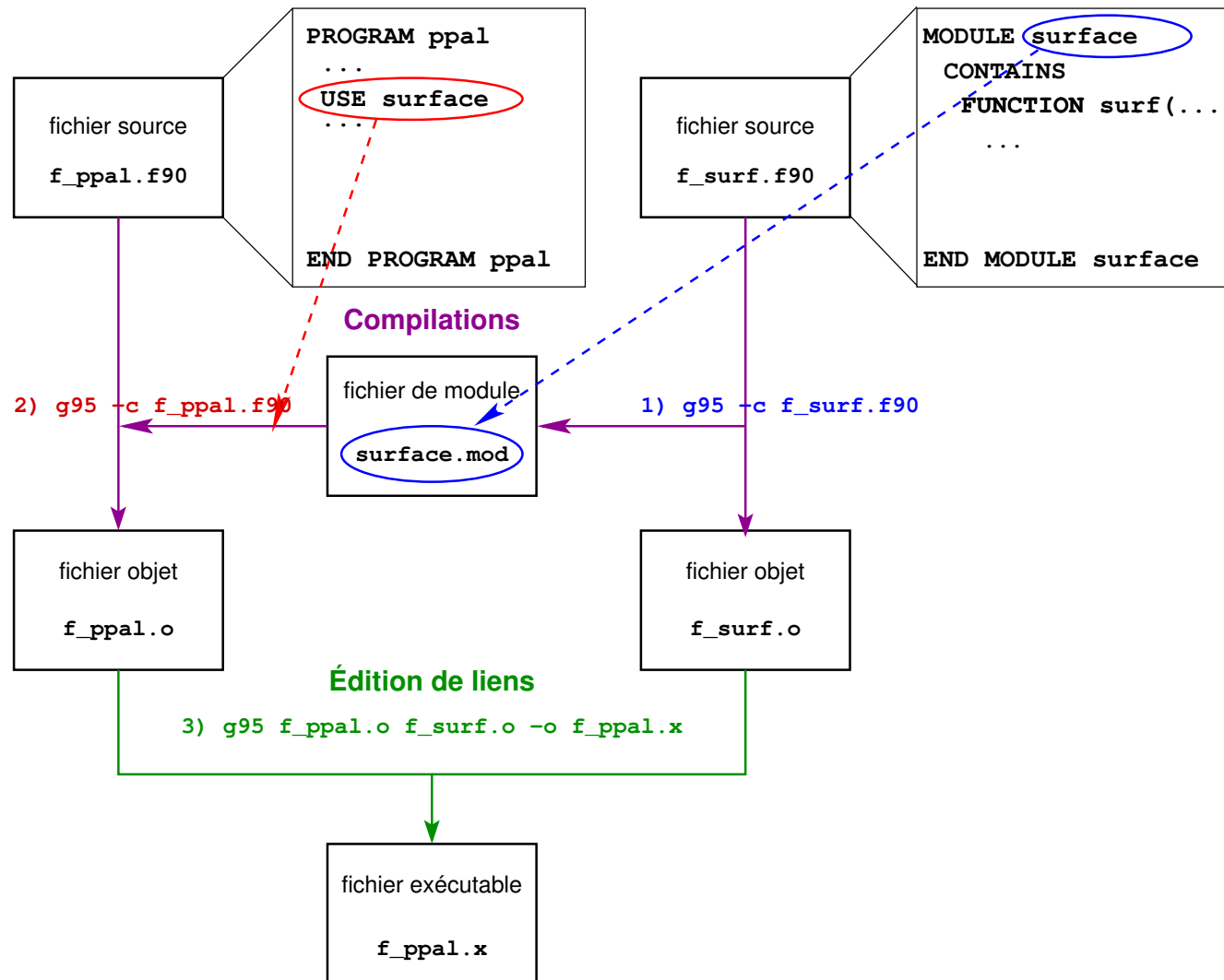
Mais risque d'incohérence entre **déclaration** et **définition** en particulier dans la phase de développement.

⇒ **Solution plus robuste**

Fortran	langage C
<p>Encapsuler les procédures dans des modules : le compilateur crée alors des fichiers .mod décrivant l'interface.</p> <p>L'instruction USE permet de relire cette interface quand on compile les appelants.</p>	<p>Déclarer les prototypes dans les fichiers d'entête * .h et les inclure à la fois :</p> <ul style="list-style-type: none"> — dans la définition — et dans les fonctions appelantes.
<p>⇒ Compiler les modules utilisés avant de compiler les appelants</p>	<p>⇒ Éviter les déclarations multiples si plusieurs inclusions</p> <pre>#ifndef ... #define ... insérer ici les prototypes #endif</pre>



Compilation séparée en C avec **fichier d'entête partagé** par appelé et appelant



Compilation séparée en fortran :

- 1) compilation du module
- 2) compilation de l'appelant
- 3) édition de liens

13.3 Fichiers d'entête (*header files*) en C

13.3.1 Définition et usage

Les fichiers d'entête peuvent contenir des déclarations de fonctions (**prototypes**) ou de nouveaux types (**struct** ou **typedef**).

Les prototypes des fonctions doivent être inclus dans :

- **le fichier où la fonction est définie**, pour assurer la cohérence déclaration/définition,
- **les fichiers où la fonction est appelée**, pour assurer la cohérence déclaration/appel.

L'inclusion se fait au moyen d'une directive préprocesseur :

```
#include "fct.h"
```

(guillemets " " pour les fonctions personnelles et non chevrons <>).

Pour les fonctions comme pour les types, il faut se protéger contre les **déclarations multiples** dans le cas d'inclusions multiples.

13.3.2 Structure d'un fichier d'entête

Soit un fichier `fct.c` contenant la définition des fonctions `produit` et `affiche`. Le fichier de déclarations `fct.h` correspondant peut s'écrire :

```
#ifndef FCT    // si FCT n'est pas defini...
#define FCT    // ... definir FCT...
double produit(double x, double y); // déclarer produit
void affiche(double res);           // déclarer affiche
#endif        // ... fin du if
```

Le test préprocesseur permet d'éviter les **déclarations multiples**.

13.4 Exemple de programme C en plusieurs fichiers

```
/* fichier main.c */
#include <stdio.h> // prototypes de printf, scanf...
#include <stdlib.h> // prototype de exit...
#include "fct.h" // prototypes de produit et affiche
int main(void) {
    double a, b, prod;
    printf("Entrer deux nombres reels:\n");
    scanf("%lg %lg", &a, &b);
    prod = produit(a,b); // appel de produit
    affiche(prod); // appel de affiche
    exit(EXIT_SUCCESS);
}
```

```
/* fichier fct.h : declaration des fonctions */  
#ifndef FCT  
#define FCT  
double produit(double x, double y);  
void affiche(double res);  
#endif /* FCT */
```

```
/* fichier fct.c : definition des fonctions */  
#include <stdio.h>  
#include <stdlib.h>  
#include "fct.h" // declaration de produit et affiche  
double produit(double x, double y) {  
    return x*y;  
}  
void affiche(double res) {  
    printf("resultat = %g\n", res);  
}
```

Structure de ce programme : 3 fichiers texte

- le fichier source principal **main.c** : déclaration et appel de deux fonctions,
- le fichier source **fct.c** : définition de deux fonctions,
- le fichier d'entête **fct.h** : déclaration de deux fonctions définies dans **fct.c** et appelées dans **main.c**

Compilation séparée :

```
gcc-mni-c99 -c fct.c
```

```
gcc-mni-c99 -c main.c
```

⇒ création de 2 fichiers objet : **main.o** et **fct.o**,

Édition de liens : en partant des fichiers objet

```
gcc-mni-c99 main.o fct.o -o main.x
```

⇒ création d'un fichier exécutable **main.x**

13.5 Bibliothèques de fichiers objets

- **Intérêt** : regrouper dans **un seul fichier** toute une collection de **fichiers objets** de procédures compilées pour simplifier les futures commandes d'édition de liens qui utilisent ces procédures.
- **Interface** : regrouper les prototypes de toutes les procédures de la bibliothèque dans un seul fichier (**.h** en C et **.mod** en fortran).

13.5.1 Bibliothèques statiques et bibliothèques dynamiques

Deux types de bibliothèques d'objets :

- **bibliothèque statique** (archive \Rightarrow extension **.a**)
édition de liens **statique** : objets intégrés à l'exécutable
 \Rightarrow exécutable autonome mais plus volumineux
- **bibliothèque dynamique partageable** (*shared object* \Rightarrow extension **.so**)
édition de liens **dynamique** : objets chargés plus tard en mémoire
lors de l'exécution \Rightarrow exécutable non autonome mais plus petit

13.5.2 Erreurs courantes lors de l'usage des bibliothèques

- En C, **le fichier d'entête** n'a pas été inclus : le prototype est inconnu.
En fortran, **le module** n'a pas été invoqué via **USE** : l'interface n'est pas connue
Erreur lors de la compilation :
attention : implicit declaration of function f1
- **La bibliothèque objet** n'a pas été indiquée (option **-l**) : le code binaire des procédures appelées n'est pas accessible
Erreur lors de l'édition de liens (appel de **ld**)
undefined reference to `f1'
collect2: ld a retourné 1 code d'état d'exécution

13.6 Commande de gestion des bibliothèques statiques : ar

Actions principales de **ar** (syntaxe proche de **tar**) :

→ **r** ajout ou **r**emplacement d'une liste de membres

```
ar rv libtab.a double1d.o double1d_libere.o
```

→ **t** liste les membres de la bibliothèque

```
ar tv libtab.a
```

```
rw-r--r-- 904/800      860 Jan 29 11:17 2008 double1d.o
```

```
rw-r--r-- 904/800      808 Jan 29 11:17 2008 double1d_libere.o
```

→ **x** **e**xtraction d'une liste de membres

```
ar xv libtab.a double1d.o
```

→ **d** destruction d'une liste de membres

```
ar dv libtab.a double1d_libere.o
```

→ **v** option (**v**erbose) avec messages d'information

→ **u** option (**u**ppdate) mise à jour seulement ⇒ **ar ruv**

→ **s** génère un index des **s**ymboles définis par les membres de l'archive

13.6.1 Création d'une bibliothèque statique (archive)

Pour créer une bibliothèque **statique** en C de nom **libtab**, il faut :

1. compiler les fichiers à insérer dans la bibliothèque

```
gcc -c double1d.c double1d_libere.c
```

2. insérer les objets dans la bibliothèque (statique) :

```
ar rv libtab.a double1d.o double1d_libere.o
```

si nécessaire : générer un index pour accélérer l'accès aux membres de la bibliothèque et leur permettre de s'appeler entre eux :

```
ar s libtab.a
```

3. créer un fichier d'entête **tab.h** contenant les prototypes des fonctions utilisées

En fortran, la compilation des modules encapsulant les procédures génère à la fois le fichier objet **.o** et le fichier de module **.mod** qui contient les interfaces (équivalent du fichier d'entête).



Les fichiers de module ne sont **pas portables** : ils dépendent du compilateur et parfois de sa version !

13.6.2 Utilisation d'une bibliothèque statique en C (entête et archive)

- 1) Lors de la **compilation**, les prototypes des fonctions sont déclarés via `#include "tab.h"` dans le code source utilisant la bibliothèque
- 2) Lors de l'**édition de liens**, le code objet des fonctions est extrait de l'archive `libtab.a` ⇒ option `-ltab` placée **après** le fichier appelant

En supposant que tous les fichiers sont dans des répertoires recherchés par défaut :

1) `gcc -c main.c` (qui relit `tab.h`)

2) `gcc main.o libtab.a`

avec le nom du fichier d'archive ou

`gcc main.o -ltab` avec l'option `-l`

En une seule commande avec compilation et édition de liens

`gcc main.c -ltab`

Si la bibliothèque est installée par l'utilisateur `user` dans des répertoires non-standard, `/home/user/include` et `/home/user/lib`, il faut compléter la liste des chemins d'accès avec les options `-I` et `-L` :

1) `-I` pour le répertoire où se trouve l'entête `tab.h`

2) `-L` pour le répertoire où se trouve l'archive `libtab.a`

`gcc -I/home/user/include -L/home/user/lib main.c -ltab`

13.6.3 Retour sur la bibliothèque standard du C

La bibliothèque standard du C est elle-même composée de sous-bibliothèques.

(les fichiers d'archive associés sont dans : `/usr/lib` ou `/lib`, ...)

À chaque sous-bibliothèque est associé un fichier d'entête :

(ces 24 fichiers sont dans : `/usr/include` ou `/include`, ...)

- **stdio.h** : prototypes de `scanf`, `printf`, `fopen`, `fclose`, ...
- **stdlib.h** : prototype d'`exit`, définition de `EXIT_SUCCESS`, `EXIT_FAILURE`, ...
- **math.h** : prototypes des fonctions mathématiques
- **tgmath.h** (en C99) : généricité des fonctions mathématiques
- **limits.h** et **float.h** : limites des entiers et des flottants
- ...

1) **Dans le code source** : le fichier d'entête est entre `<...>`

2) **Édition de liens** : chargement automatique pour toutes les sous-bibliothèques

sauf la sous-bibliothèque mathématique : \Rightarrow utiliser l'option `-lm` de `gcc`.

13.6.4 Retour sur la bibliothèque `libmnitab`

1. Le prototype des fonctions de cette bibliothèque est dans le fichier `mnitab.h`

Dans le code source : `#include "mnitab.h"`

2. Le fichier d'archive associé à cette bibliothèque est : `libmnitab.a`

Édition de liens : ajouter l'option `-lmnitab` après les objets

Pour simplifier : à l'UPMC, utiliser `gcc+mni` ou `gcc+mni-c99`


qui complètent les chemins de recherche des entêtes et des bibliothèques

⇒ `gcc+mni` est un alias vers :

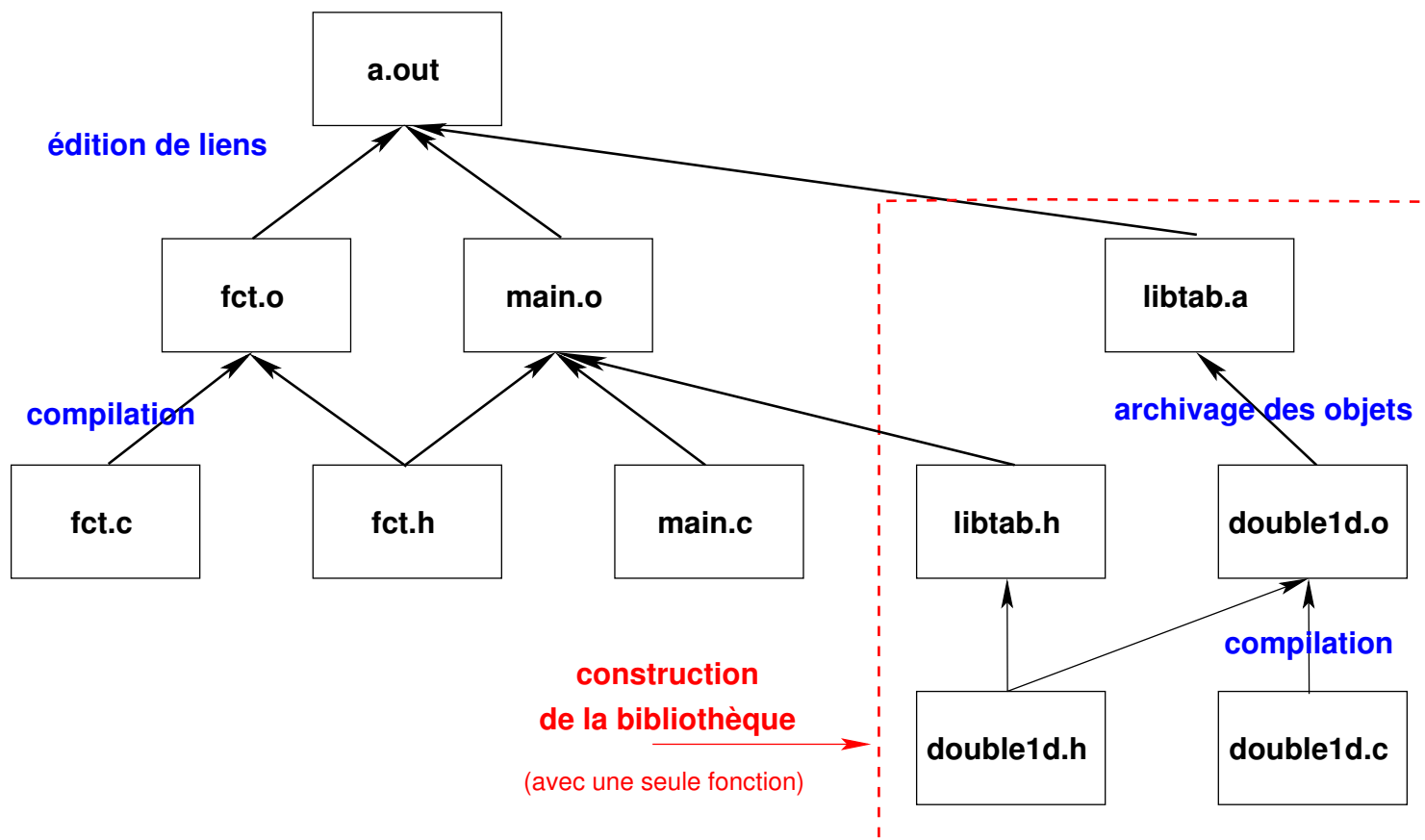
```
gcc-mni -I/home/lefrere/M1/mdv2010-32/include \
        -L/home/lefrere/M1/mdv2010-32/lib
```

où `/home/lefrere/M1/mdv2010-32/include/` contient `mnitab.h`

et `/home/lefrere/M1/mdv2010-32/lib/` contient `libmnitab.a`

 Les objets ne sont pas portables ⇒ **version 64 bits** pour nouvel `sappli1` dans `/home/lefrere/M1/sappli1-64/lib/` et idem pour `include/` et alias `gcc64+mni` avec options `-L` et `-I` associées

13.6.5 Bilan sur la création et l'usage d'une bibliothèque statique



Remarque : la création d'une bibliothèque n'a d'intérêt que si elle contient de nombreuses fonctions que l'on n'a pas besoin de modifier. Sinon, préférer une compilation en fichiers séparés sans bibliothèque ou l'emploi de l'utilitaire **make**.

13.7 Génération d'un fichier exécutable avec `make`

13.7.1 Principe

La commande **make** permet d'**automatiser** la génération d'un fichier (souvent exécutable) ou **cible** (**target**) qui **dépend** d'autres fichiers en mettant en œuvre certaines **règles** (**rules**) de construction décrites dans un fichier **makefile**. **make** minimise les opérations de mise à jour en s'appuyant sur les règles de dépendance et les **dates** de modification des fichiers.

Application la plus classique :

reconstituer automatiquement un programme exécutable à partir des fichiers sources en ne recompilant que ceux qui ont été modifiés.

- **cible** (**target**) : en général un fichier à produire
- **règle** de production (**rule**) : liste des commandes à exécuter pour construire une cible (compilation pour les fichiers objets, édition de lien pour l'exécutable)
- **dépendance** : ensemble des fichiers nécessaires à la production d'une cible

13.7.2 Construction d'un makefile

Le fichier **makefile** liste les cibles, décrit les dépendances et les règles. Mais des **règles implicites** permettent d'automatiser la création des cibles les plus classiques. On peut enrichir ces méthodes s'appuyant sur les suffixes des fichiers.

Syntaxe des dépendances et des règles :

```
cible: liste des dépendances
(tabulation) règle de construction (en shell)
```

⇒ nécessite d'intégrer des commandes shell dans le **makefile**

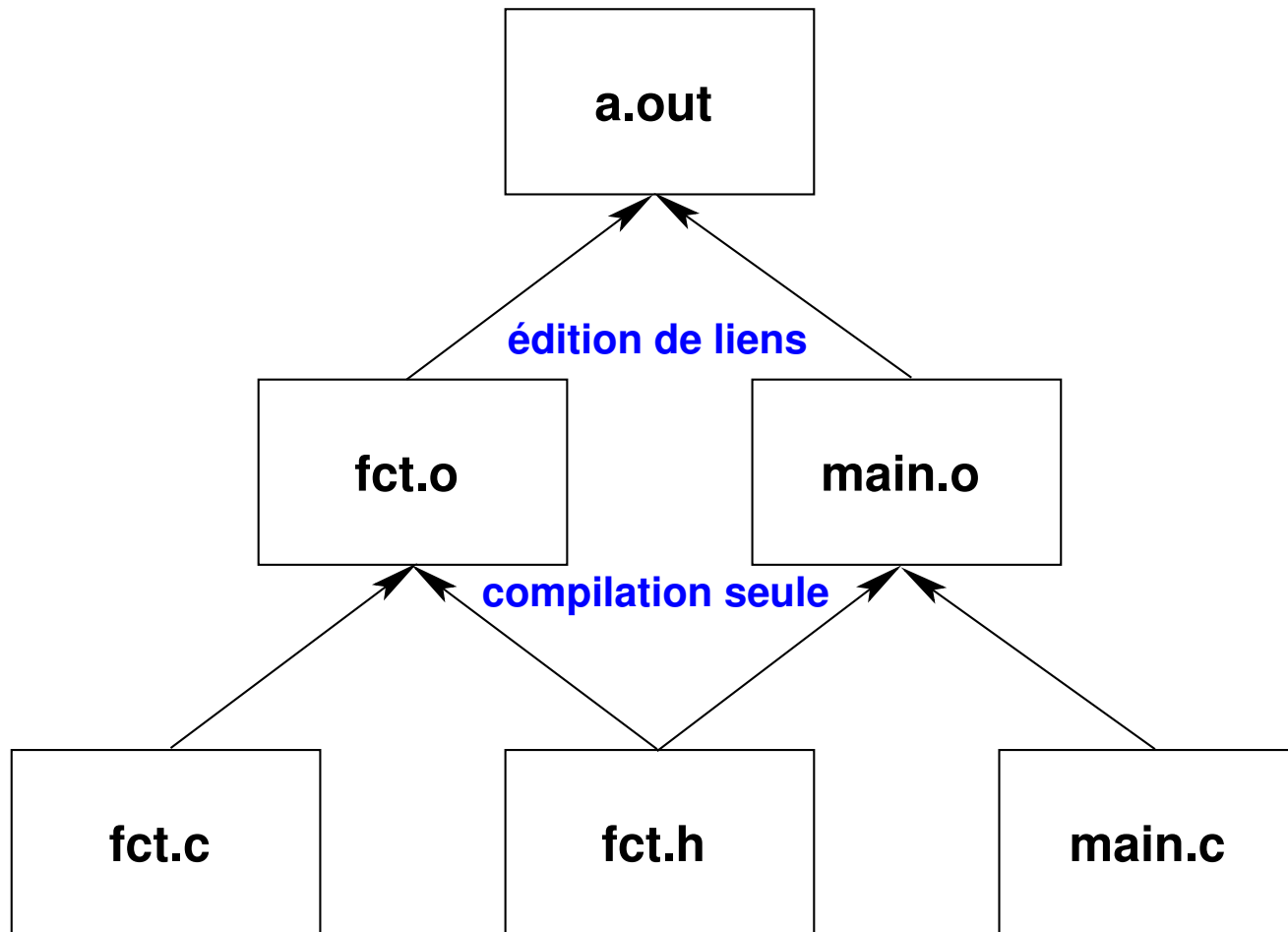
Le fichier **makefile** est construit à partir de l'arbre des dépendances.

```
gcc -MM fichier.c affiche les dépendances de fichier.o
                                                    (nécessite les .h)
```

```
g95 -M fichier.f90 affiche les dépendances de fichier.o
                                                    (nécessite les .mod)
```

13.7.3 Exemple élémentaire de `makefile` en C

Arbre des dépendances (exploré **récurivement** par `make`)



```
## fichier makefile construit a partir  
## de l'arbre des dependances
```

```
# première cible = exécutable => règle = édition de liens
```

```
a.out : fct.o main.o  
___TAB___gcc fct.o main.o
```

```
# cibles des objets => règle = compilation seule avec gcc -c
```

```
fct.o : fct.c fct.h  
___TAB___gcc -c fct.c  
main.o : main.c fct.h  
___TAB___gcc -c main.c
```

```
# ménage : suppression des fichiers restructuribles
```

```
clean:  
___TAB___/bin/rm -f a.out *.o
```


13.7.4 Utilisation d'un `makefile`

`make cible`

lance la production de la *cible* en exploitant le fichier **`makefile`** du répertoire courant.

`make -n cible`

affiche les commandes que devrait lancer `make` pour produire la *cible*

`make -f mymakefile cible`

pour utiliser ***`mymakefile`*** au lieu de `makefile`

En savoir plus :

<http://www.gnu.org/software/make/manual/make.html>

Remarque : `cmake` est un utilitaire équivalent à `make` et adapté à un ensemble plus large de systèmes d'exploitation (dont Windows).

Voir : <http://www.cmake.org/>

En guise de conclusion

Fortran et C : deux langages de haut niveau finalement assez proches

Différences fortes apparues avec fortran 90 au niveau des tableaux
mais tableaux automatiques en C99

Fortran et C sont maintenant **interopérables** de façon standard grâce aux outils
fortran de la norme 2003 : on peut appeler du C à partir du fortran et l'inverse.

Évolutions vers le **langage objet** avec **fortran 2003** d'un côté et **C++** de l'autre.