UPMC

Master P&A/SDUEE

UE MNI (MU4PY109)

Méthodes Numériques et Informatiques

Fortran 95/2003/2008 et C

Sofian.Teber@lpthe.jussieu.fr
Jacques.Lefrere@upmc.fr
Albert Hertzog

2019–2020

1 Introduction

1.1 Langage compilé et langage interprété

Langage **compilé**	Langage **interprété**
C, C++, **fortran**	**shell**, **python**, perl, php, scilab, awk
Le compilateur analyse l'ensemble du programme avant de le traduire en langage machine une fois pour toutes. $\Rightarrow$ **optimisation** possible L'exécutable produit est **autonome** mais dépend du processeur	**L'interpréteur** exécute le code source (**script**) instruction par instruction. $\Rightarrow$ pas d'optimisation L'interpréteur est nécessaire à chaque exécution.
En cas d'erreur de syntaxe	
le binaire exécutable n'est pas produit, donc pas d'exécution !	les instructions avant l'erreur sont exécutées
Compilé + interprété, par exemple **java** compilation $\rightarrow$ bytecode portable interprété par JVM (machine virtuelle java)	

1.2 Étapes de la programmation en langage compilé

Les **étapes** de la programmation (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, et utilisant des bibliothèques : C, fortran, ...

⇒ code **source** : fichier texte avec instructions commentées

compréhensible pour le concepteur... et les autres

⚠ Seul le fichier **source** est **portable** (indépendant de la machine)

— **compilation** : transformer le code source en un code machine

⇒ code **objet**, puis code **exécutable** : fichiers binaires

— **exécution** : tester le bon fonctionnement du programme

⇒ exploitation du code et production des résultats

1.3 Compilation et édition de liens

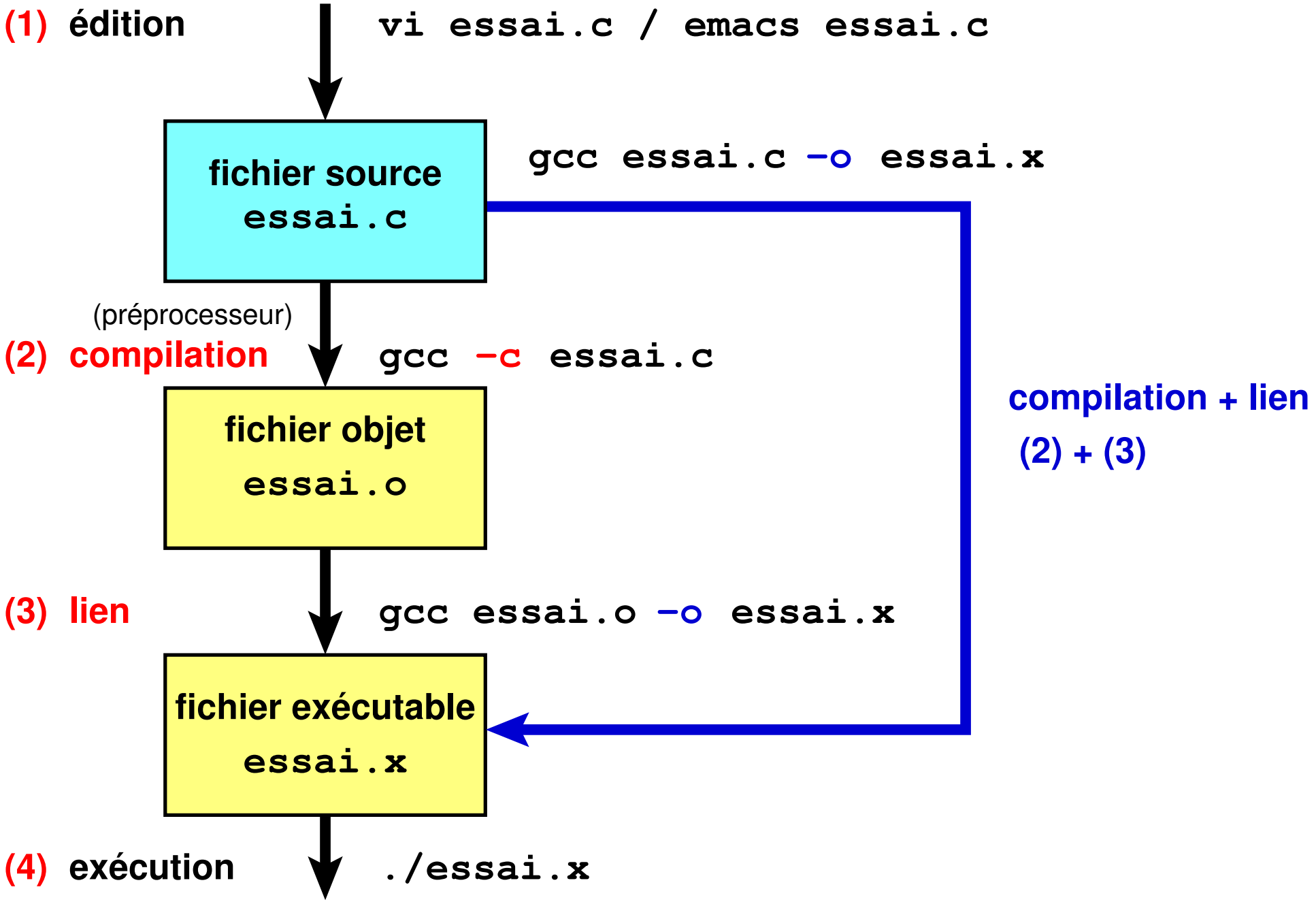
— Fichier **source** (texte) de suffixe `.c` en C, `.f90` en fortran 90
écrit au moyen d'un **éditeur de texte**

— Fichier **objet** (binaire) de suffixe `.o` :
code machine généré par le **compilateur**

— Fichier **exécutable** (binaire) `a.out` par défaut
produit par l'**éditeur de liens**

La commande de compilation `gcc essai.c` ou `gfortran essai.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 $\rightarrow$ fichier objet `essai.o`
3. **édition de liens** (*link*) : le compilateur lance `ld` $\rightarrow$ fichier exécutable `a.out`
   assemblage des codes objets et résolution des appels aux bibliothèques


(1) édition
vi essai.c / emacs essai.c
fichier source
essai.c
gcc essai.c -o essai.x
(préprocesseur)
(2) compilation
gcc -c essai.c
fichier objet
essai.o
compilation + lien
(2) + (3)
(3) lien
gcc essai.o -o essai.x
fichier exécutable
essai.x
(4) exécution
./essai.x

Rôle du compilateur

— analyser le code source,

— signaler les erreurs de syntaxe,

— produire des avertissements sur les constructions suspectes,

— convertir un code source en code machine (sauf si erreur de syntaxe !),

— optimiser le code machine.

⇒ **faire du compilateur un assistant efficace** pour anticiper les problèmes avant des erreurs à l'édition de liens ou, pire, à l'exécution.

	C	fortran
GNU Compiler Collection	**`gcc`**	**`gfortran`**
Documentation	`http://gcc.gnu.org`	
Intel	`icc`	`ifort`

Principales options de compilation

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

→ `-c` : préprocesseur et compilation seulement (produit l'objet)

→ `-o essai.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 utiles à la **mise au point** :

→ conformité aux standards du langage : `-std=c99` ou `-std=f2003`

→ avertissements (*warnings*) sur les instructions suspectes (variables non utilisées, instructions apparemment inutiles, conversions de type, ...) : `-Wall`

→ vérification des passages de paramètres des procédures (nécessite un contrôle interprocédural, donc les prototypes ou interfaces)

alias avec options sévères et précision du standard	
`gcc-mni-c89`	`gfortran-mni`
`gcc-mni-c99`	`gfortran2003-mni`

1.4 Historique

1.4.1 Langage fortran : Fortran = Formula Translation

— 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...

$\Rightarrow$ ne plus écrire de fortran 77

— 1997 : fortran 95 = mise à jour mineure

— 2004 : adoption du standard **fortran 2003**

nouveautés : interopérabilité avec C, arithmétique IEEE, accès au système, allocations dynamiques étendues, aspects objet...

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

— 2010 (20 sept) : adoption du standard fortran 2008 (`https://wg5-fortran.org/`)

— 2018 (nov) : publication officielle du fortran 2018 (ex-standard 2015)

— 2019 : fortran 2008 complet sur Cray et Intel, discussions sur fortran 202x

1.4.2 Langage C

— langage conçu dans les années 1970

— 1978 : **The C Programming Langage** 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**
nouveaux types (booléen, complexe, entiers de diverses tailles (prise en compte des processeurs 64 bits), caractères larges (unicode), ...),
généricité dans les fonctions numériques,
déclarations tardives des variables, tableaux automatiques de taille variable...

— norme **C11** (ex-C1x) parue en avril 2011

— norme **C18** (ex-C17), révision mineure approuvée en juin 2017, parue en 2018

— base d'autres langages dont le **C++** (1er standard en 1998, C++17 en avril 2017)
puis java, php, ...

1.5 Intérêts respectifs du C et du fortran

Langage fortran	Langage C
codes portables et pérennes grâce aux **normes** du langage et des bibliothèques	
langages de **haut niveau** structures de contrôle, structures de données, fonctions, compilation séparée, ...	
pas d'accès au bas niveau (adresses par ex.)	mais aussi ... langage de **bas niveau** (manipulation de bits, d'adresses, ...)
langage puissant, efficace	
mais aussi ... peu permissif	mais aussi ... permissif !
spécialisé pour le **calcul scientifique** généricité des fonctions mathématiques	plus généraliste ex. : écriture de systèmes d'exploitation
tableaux multidimensionnels et fonctions associées (cf. matlab et numpy)	

Exemple : appel à la procédure **geev** 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 spécifique : type `float` seulement

Langage Fortran 95

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

5 arguments + status optionnel passé par mot-clef générique

1.6 Format des instructions

langage C		langage Fortran
libre	format du code	libre du fortran 90 fixe en fortran 77
⇒ mettre en évidence les structures par la mise en page (indentation)		
instruction simple **;** instruction composée **}**	une instruction se termine par	la fin de la ligne sauf si **&** à la fin
entre **/*** et ***/** peut s'étendre sur plusieurs lignes	délimiteurs de commentaire	
// en C99 et C++	introducteur de commentaire ⟶ fin de la ligne	**!**
⚠ distinction	maj/minuscule	pas de distinction
lignes de directives pour le préprocesseur : **#** en première colonne		

1.7 Exemple de programme C avec une seule fonction utilisateur : `main`

```c
#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       */
```

1.8 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
END PROGRAM ppal                              ! << fin du programme ppal
```

2 Types et déclarations des variables

2.1 Représentation des nombres : domaine (*range*) et précision

Typage statique des variables $\Rightarrow$ nombre de bits fixe $\Rightarrow$ **domaine couvert limité**

Mais distinguer :

— les **entiers représentés exactement**

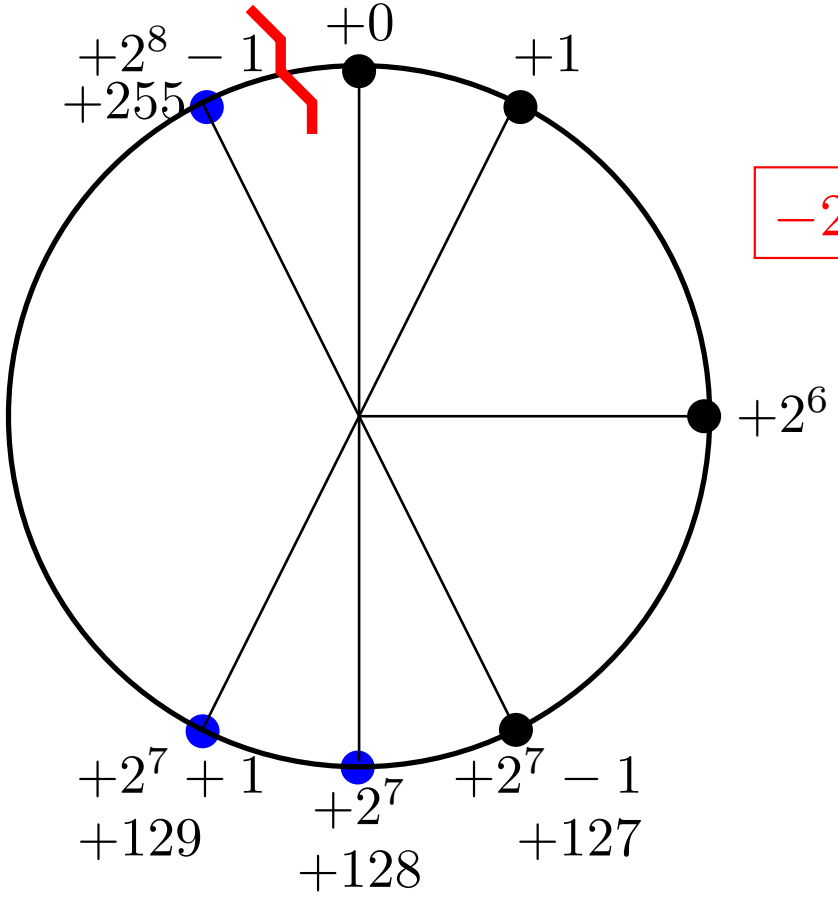
— les **réels représentés approximativement** en virgule flottante

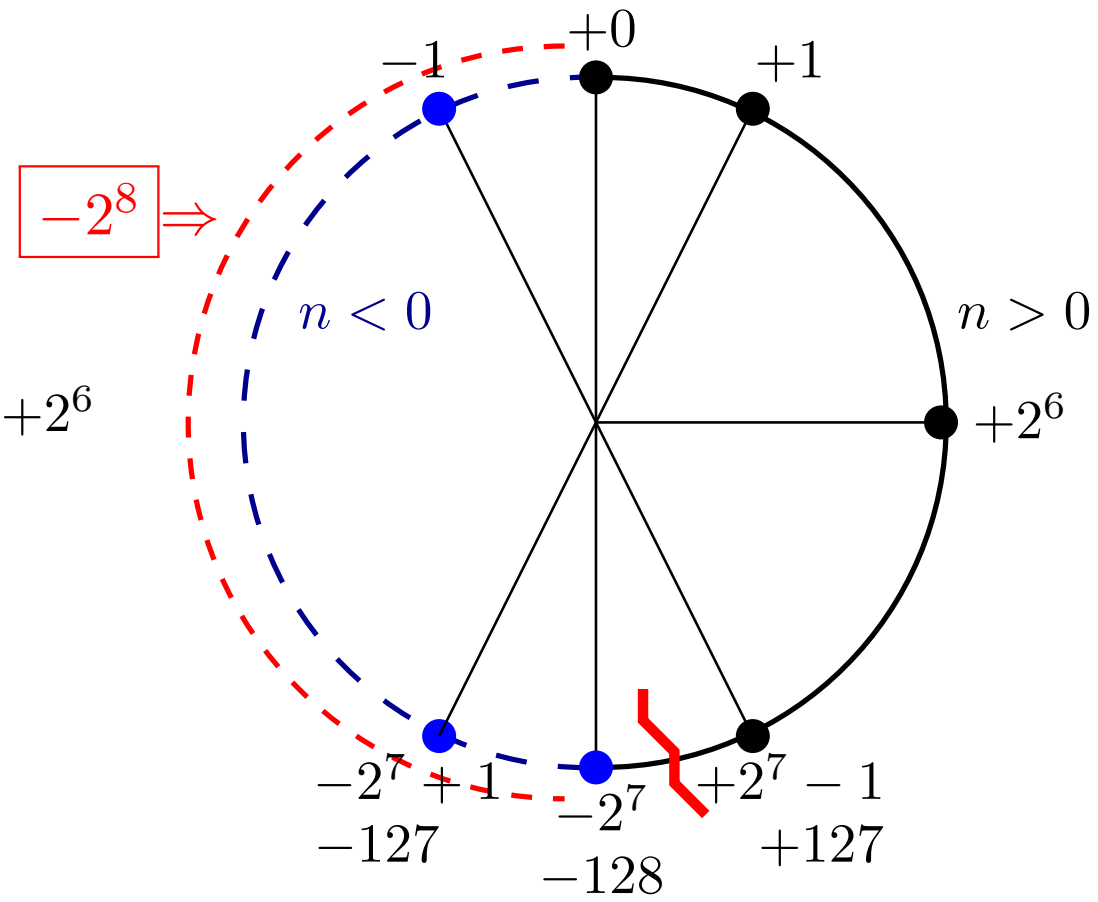
⚠ $\Rightarrow$ ne jamais compter avec des réels

2.1.1 Domaine des entiers signés

Exemple introductif des entiers sur **1 octet** (8 bits) : $2^8 = 256$ valeurs possibles

taille	entiers non signés	entiers signés
1 octet	$0 \rightarrow 255 = 2^8 - 1$	$-2^7 = -128 \rightarrow +127 = 2^7 - 1$





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.

2.1.2 Limites des entiers signés sur 32 et 64 bits

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

		fortran	C
	valeur max	fonction HUGE	`/usr/include/`**`limits.h`**
sur 32 bits = 4 octets	$2^{31} \approx 2 \times 10^9$	**`HUGE(1)`**	**`INT_MAX`**
sur 64 bits = 8 octets	$2^{63} \approx 9 \times 10^{18}$	**`HUGE(1_8)`**	**`LLONG_MAX`**

$\rightarrow$ en fortran, choix des variantes (**`KIND`**) d'entiers selon le domaine (*range*) par la fonction **`SELECTED_INT_KIND(r)`** pour $|i| \leqslant 10^r$

$\rightarrow$ en C89, les tailles des entiers dépendent du processeur (non portable)

$\rightarrow$ C99 : types entiers étendus à nb d'octets imposé : **`int32_t`**, **`int64_t`** ...

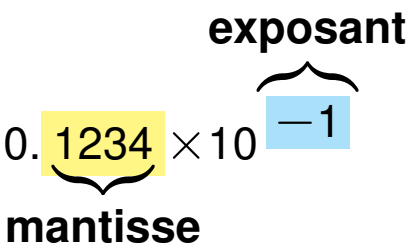
⚠ Dépassement de capacité en entier positif $\Rightarrow$ perte de retenue et passage en négatif

2.1.3 Domaine et précision des réels flottants

Représentation approchée en **virgule flottante** pour concilier
dynamique et précision relative presque constante
Par exemple en base 10, avec 4 chiffres après la virgule, comparer les représentations approchées (par troncature) en virgule fixe et flottante :

nombre exact	virgule fixe	virgule flottante
	par troncature	
0.0000123456789	⚠ .0000	0.1234×10^{-4}
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}

Virgule flottante
en base 10



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

— m bits de mantisse $\Rightarrow$ **précision** limitée

— q bits de l'exposant $\Rightarrow$ **domaine** fini

Ajouter 1 bit de signe $\Rightarrow$ nombre de bits $= m + q + 1$

— À exposant (puissance de 2) fixé : progression arithmétique dans chaque octave

$\Rightarrow 2^m$ valeurs par octave

ε = la plus petite valeur telle que $1 + \varepsilon > 1$ donc $1 + \varepsilon =$ successeur de 1

ε est le pas des flottants dans l'octave $[1, 2[\Rightarrow \boldsymbol{\varepsilon = 1/2^m}$

Précision relative $\leqslant \boldsymbol{\varepsilon = 1/2^m}$

Flottants sur 32 bits : $m =$ **23 bits de mantisse** $\Rightarrow \varepsilon = \boldsymbol{2^{-23}} \approx \boldsymbol{10^{-6}/8}$

— Octaves en progression géométrique de raison 2

q bits d'exposant $\Rightarrow 2^q - 2$ octaves (+ codes non numériques)

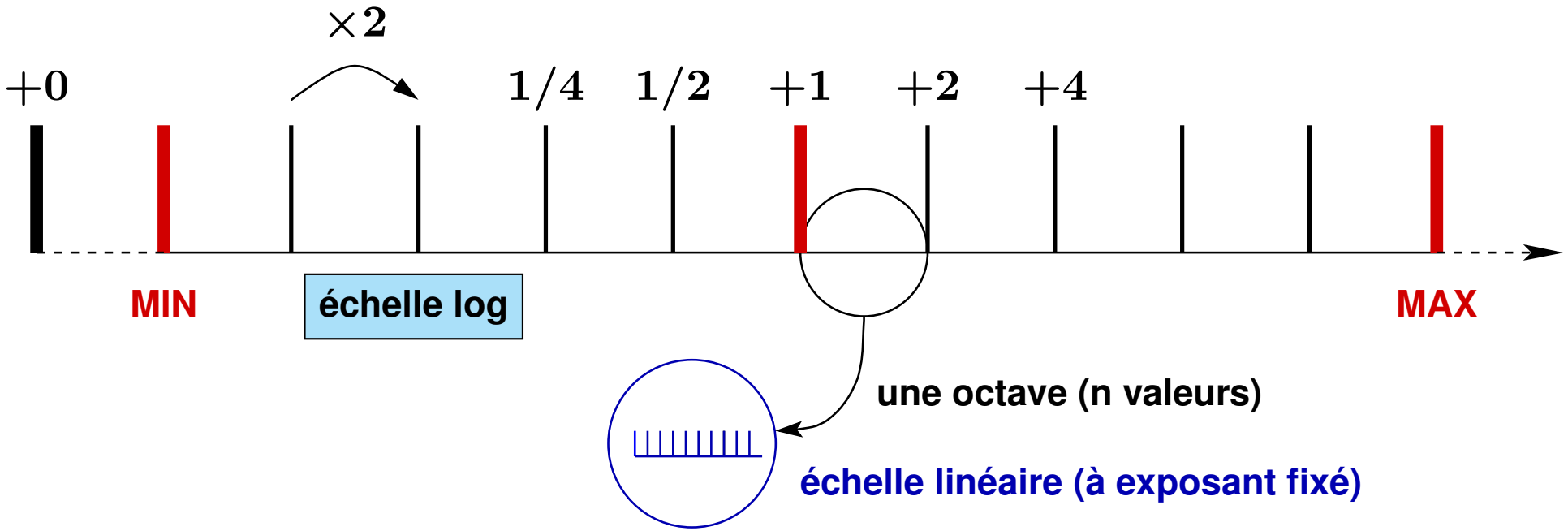
Flottants sur 32 bits : $q =$ **8 bits d'exposant** donc 254 octaves

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

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.

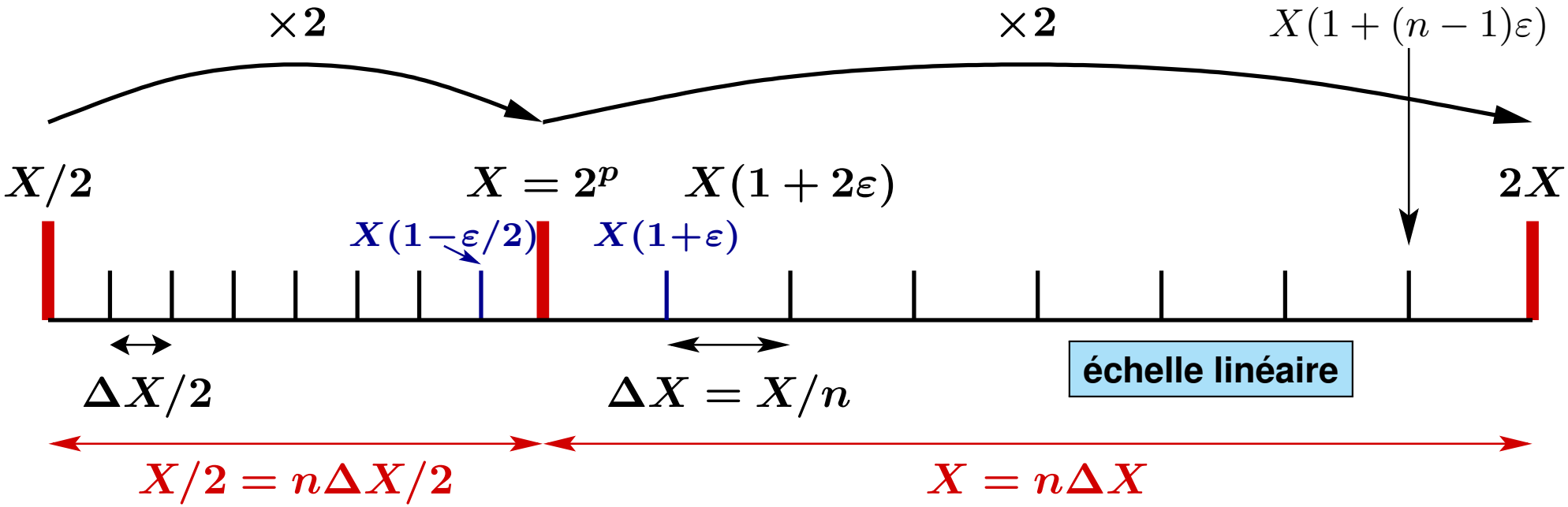


Domaine des flottants positifs normalisés (**échelle log**)

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

2.1.4 Caractéristiques numériques des flottants sur 32 et 64 bits

nb total	mantisse	exposant
32 bits	23 bits	8 bits
64 bits	52 bits	11 bits

(voir norme IEEE 754-2008)

	fortran	C (`float.h`)	valeur
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(1d0)`**	**`DBL_EPSILON`**	$2^{-52} \approx 2,2 \times 10^{-16}$

2.1.5 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 limitant le domaine de x (la plus petite des 2 valeurs absolues des exposants extrêmes)
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

$\Rightarrow$ demander la variante (**KIND**) du type numérique suffisante

`ki = SELECTED_INT_KIND(r)` pour les entiers allant jusqu'à 10^r

`kr = 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)

2.2 Types de base

Types représentés exactement

langage C	Type	fortran 90
////C99 **`bool`**	booléen	**`logical`**
`char`	caractère	≈ `character(len=1)`
////(tableau de `char`)	chaîne de caractères	**`character(len=....)`**
`short int`	entier court	`integer(kind=2)`
`int`	entier par défaut	**`integer`**
`long int`	entier long	`integer(kind=4/8)`
C99 **`long long`**	entier long long	`integer(kind=8)`

Types représentés approximativement

en virgule flottante : mantisse et exposant

langage C	Type	fortran 90
`float`	réel simple précision (32 bits)	**`real`**
`double`	double précision (64 bits)	**`double precision`**
`long double`	précision étendue ($\geqslant$ 80 bits)	`real(kind=10/16)`
C99 ou **`<complex.h>`**	complexe	**`complex`**
`float complex`		
`double complex`	+ variantes de précision	`complex(kind=?)`
`long double complex`		

2.3 Les constantes

langage C C99	Type	fortran 90
///// C99 : **true** ($\neq 0$) **false** (0)	booléen	.TRUE. .FALSE.
'a'	caractère	'a' "a"
"chaine" "s'il"	chaîne	'chaine' "s'il"
17	entier court	17_**2**
17	**entier décimal**	17
⚠ **0**21 (attention)	entier 17_{10} en **o**ctal	**O**'21'
0x11	entier en hexadécimal	**Z**'11'
17**L**	entier long	17_**8**

langage C	C99	Type	fortran 90	
`-47.1f`	`-6.2e-2f`	réel simple précision	**`-47.1`**	**`-6.2e-2`**
`-47.1`	**`-6.2e-2`**	double précision	`47.1_8`	`-6.2e-2_8`
`-47.1L`	`-6.2e-2L`	double précision long	`47.1_16`	`-6.2e-2_16`
///// sauf **`I`** C99 :	`2.3-I*.5`	complexe	`(2.3, -5.)`	

⚠ Erreur fréquente : utiliser une expression à la place d'une constante.

	préférer	éviter	
valeur à coder	constante	expression C	expression fortran
$1{,}5 \times 10^{3}$	**`1.5e3`**	**`1.5*pow(10,3)`**	**`1.5*10**3`**
$1{,}5 \times 10^{-3}$	**`1.5e-3`**	**`1.5*pow(10,-3)`**	~~**`1.5*10**(-3)`**~~
		arguments de `pow` convertis	donne 0 (division entière)
		`1.5*pow(10.,-3)`	**`1.5*10.**(-3)`**

2.4 Déclarations des variables

Typage statique $\Rightarrow$ déclarer le type de chaque variable (fixé une fois pour toutes)

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

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

(correspondance 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 = lui affecter une valeur lors de la réservation de la mémoire

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

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

langage C $\geqslant 89$	**fortran** $\geqslant 90$
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

3 Opérateurs

3.1 Opérateur d'affectation

⚠ L'affectation **=** peut provoquer une conversion **implicite** de type du membre de droite !

⇒ 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
`entier = (int) flottant;`	`entier = INT(flottant)`

extrait de code C

```c
int n = 123456789;          // n exact (9 chiffres)
float b = 0.123456789f;     // b approché (7 chiffres)
float nf;
printf("float: %d octets \t int: %d octets\n",
        (int) sizeof(float), (int) sizeof(int));
nf = (float) n;             // conversion => à 10^(-7) près
printf("n (int)   =  %d \n" "nf (float) = %.10g \n"
        "b (float) =%.10g\n", n, nf, b);
```

```
float: 4 octets          int: 4 octets
n (int)    =  123456789  exact (entier)
nf (float) = 123456792   approché (flottant)
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 ⇒ même type pour les opérandes (sauf réel`**`entier fortran)

⚠ Division entière : `3/2` ⇝`1` (cf opérateur `//` de python 3), mais `3./2.` ⇝`1.5`

Types différents ⇒ **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
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 le type booléen (`bool`) existe en C99, avec `stdbool.h`.

3.5 Incrémentation et décrémentation en C

— **post-incrémentation** et **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** et **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 !

3.6 Opérateurs d'affectation composée en C

lvalue opérateur = expression $\Rightarrow$ **lvalue = lvalue 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) ? a : b` affecte le max de `a` et `b` à `c`

3.8 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 (ex. : dans `calloc`).

`sizeof` identificateur

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

`sizeof`(type)

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

3.9 Opérateur séquentiel «,» en C

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

⚠ Ne présente un intérêt que si **`expr1`** a un effet de bord (ex. : **`i++,j++`**)

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

3.10 Opérateurs `&` et `*` en C

&objet ⇒ adresse de l'objet

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

3.11 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 `,`

$\Rightarrow$ indiquer les priorités avec des parenthèses !

4 Entrées et sorties standard élémentaires

C	fortran 90
écriture sur **stdout** = écran	
`printf("format", liste d'expressions);`	`WRITE(*, *) &` `liste d'expressions`
lecture depuis **stdin** = clavier	
⚠ `scanf("format", liste de pointeurs);`	`READ(*, *) &` `liste de variables`
format	
format `%d`, `%g` ou `%s` ... un par variable selon le type (gabarit optionnel)	format **libre** (`*`) le plus simple mais on peut préciser
pour changer de ligne	
⚠ spécifier `\n` en sortie	forcer par `/` dans le format **changement d'enregistrement** par défaut 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 2003
entiers		
décimal	%w[.p]**d**	**I**w[.p]
	%**d**	**I0**
réels		
virgule fixe	%w[.p]**f**	**F**w.p
virgule flottante	%w[.p]**e**	**E**w.p
préférer ⇒ général	%w[.p]**g**	**G**w.p
caractères		
caractères	%w[.p]**c**	**A**[w]
chaîne	%w[.p]**s**	**A**[w]

```c
#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'addresse */
printf("La valeur de i est %d\n", i);
printf("Entrer un float, un double \n");
scanf("%g %lg", &x, &y); /* passer les adresses */
printf("x = %g et y = %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`
`long`	`%ld`
`long long`	`%lld`
`float`(convertis en)/**`double`**	(`%e`, `%f`) `%g`
`long double`	(`%Le`, `%Lf`) `%Lg`

En entrée avec `scanf` (un format par type)

Type	Format
`char`	`%c`
`short`	`%hd`
`int`	`%d`
`long`	`%ld`
`long long`	`%lld`
`float`	(`%e`, `%f`) `%g`
`double`	(`%le`, `%lf`) `%lg`
`long double`	(`%Le`, `%Lf`) `%Lg`

5 Structures de contrôle

Par défaut, exécution séquentielle des instructions une seule fois,
dans l'ordre spécifié par le programme.
$\Rightarrow$ trop restrictif

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

— exécution conditionnelle (**`if`** / **`else`**)
ou aiguillage (**`case`**) dans les instructions

— itération de certains blocs (**`for`**, **`do`**, **`while`**...)

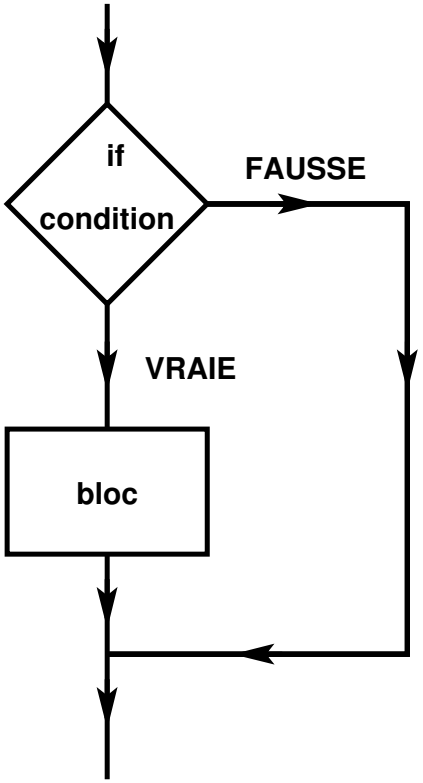
— branchements (**`cycle`** ou **`continue`**, **`exit`** ou **`break`**, ...)

$\Rightarrow$ 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`

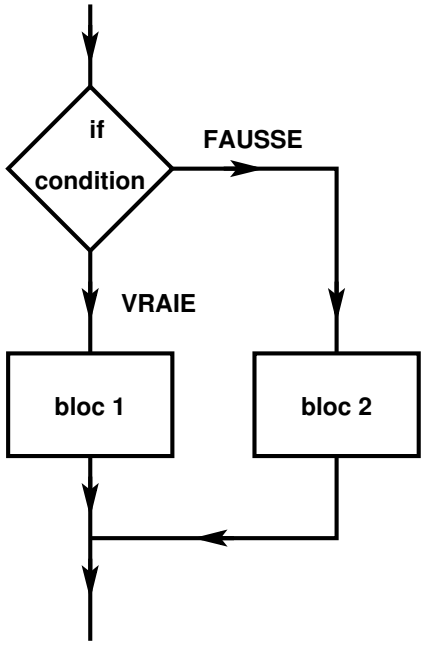


c	fortran 90
if (expression**)** instruction	**if (**expr. log.**)** instruction

if (expression**) {**
bloc d'instructions
}
	if (expr. log.**) then**
	bloc d'instructions
	end if

expression testée	
entier vrai si non nul en C89	de type **booléen**

5.1.2 Alternative `if` ... `else`



c	fortran 90

if (expression**) {**
bloc d'instructions 1
}
else {
bloc d'instructions 2
}
	if (expr. log.**) then**
	bloc d'instructions 1
	else
	bloc d'instructions 2
	end if

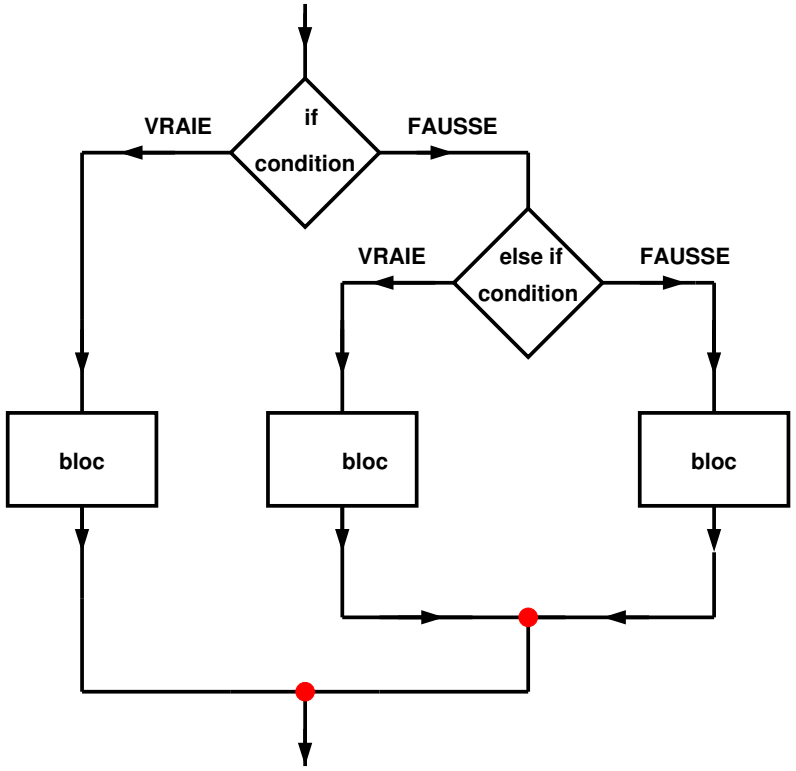
5.1.3 Exemples d'alternative if ... else

```c
#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  {
   max = j ;                /* instruction simple  */
 }
 printf(" i= %d, j= %d, max = %d\n", i, j, max);
 exit(EXIT_SUCCESS) ;
}
```

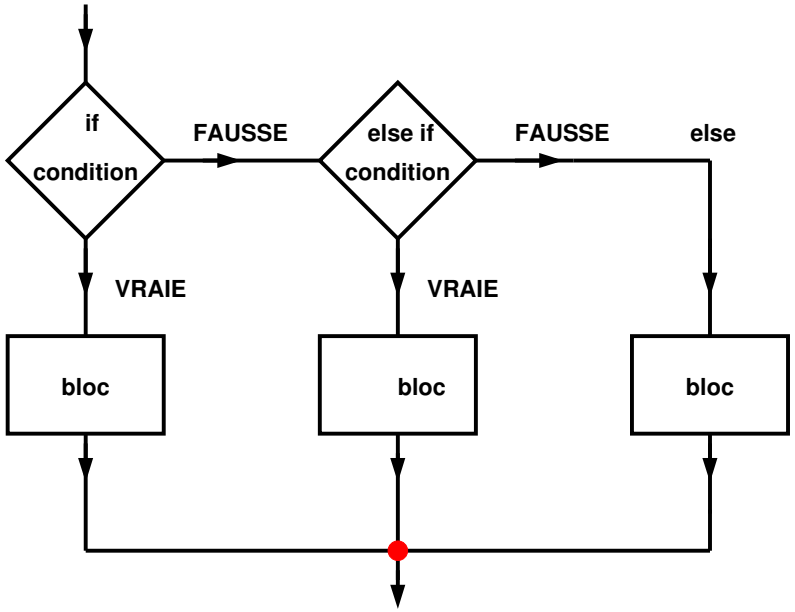
```
! structure   if then ... else ... endif
! affichage du max de deux nombres
PROGRAM alternative
IMPLICIT NONE
INTEGER :: i, j, maxij
WRITE(*,*) "entrer i et j (entiers)"
READ(*,*) i, j
IF (i >= j) THEN
  WRITE(*,*) "i >= j "
  maxij = i                    ! bloc d'instructions
ELSE
  maxij = j                    ! 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 niveaux



Fusion des retours $\Rightarrow$ un niveau



5.1.5 Aplatissement de l'imbrication avec `else if` en fortran

Structures imbriquées ⇒ deux **`end if`**

```
! deux if imbriqués
IF (i < -10) THEN ! externe
  WRITE(*,*) "i < -10"
ELSE
  IF (i < 10) THEN ! interne
    WRITE(*,*) "-10 <= i < 10"
  ELSE
    WRITE(*,*) "i >= 10 "
  END IF ! end if interne
END IF ! end if externe
```

Structure aplatie ⇒ un **`end if`**

```
! structure avec ELSE IF
IF (i < -10) THEN
  WRITE(*,*) "i < -10"
! ELSEIF sur une même ligne
ELSE IF (i < 10) THEN
  WRITE(*,*) "-10<=i<10"
ELSE
  WRITE(*,*) "i >= 10 "
END IF ! un seul END IF
```

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

c	fortran 90

switch (expr. entière**) {**
  case sélecteur1 **:**
  bloc d'instructions
  [**break ;**]
  case sélecteur2 **:**
  bloc d'instructions
  [**break ;**]
  ...
  default :
  bloc d'instructions
}
	select case (expr.**)**
	  case (sélecteur1 **)**
	  bloc d'instructions
	  case (sélecteur2**)**
	  bloc d'instructions
	  ...
	  case default
	  bloc d'instructions
	end select

sélecteur	
expression constante entière ou caractère	expression constante entière ou caractère ou liste ou intervalle fini ou semi-infini,
⚠ Sans **break**, on exécute toutes les instructions qui suivent le premier sélecteur vrai !	

5.2.1 Exemples d'aiguillage `case`

case.c

```
switch (i) /* i entier */
  {                                  /* début de bloc */
  case 0 :
    printf(" i vaut 0 \n") ;
    break;                         /* necessaire ici ! */
  case 1 :
    printf(" i vaut 1 \n") ;
    break;                         /* necessaire ici ! */
  default :
    printf(" i différent de 0 et de 1 \n") ;
  }                                  /* fin de bloc */
```

case.f90

```
SELECT CASE(i)  ! i entier              ! 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
```

case1.c

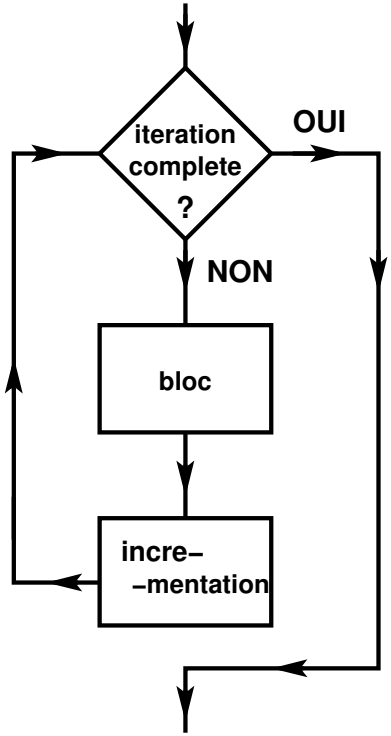
```c
/* structure case sans break
 * pour "factoriser des cas"
   et les traiter en commun */
switch (c)  /* c de type char */
  {
  case '?' :
  case '!' :
  case ';' :
  case ':' :
    printf(" ponctuation double \n") ;
    break ;        /* à la fin des 4 cas */
  default :
    printf(" autre caractère \n") ;
  }
```

case1.f90

```
! select case avec des listes de constantes
! pour les cas à traiter en commun
SELECT CASE (c) ! de type CHARACTER(len=1)
  CASE ('?','!',';',':')
    WRITE(*,*) "ponctuation double"
  CASE default
    WRITE(*,*) "autre caractère "
END SELECT
```

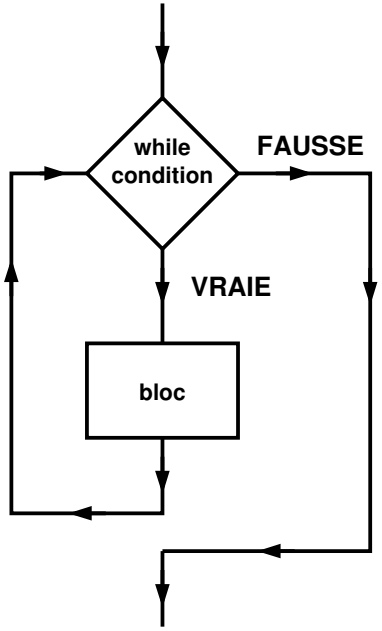
5.3 Structures itératives ou boucles

Choix selon que le nombre d'itérations est calculable avant ou non :



nombre d'itérations **connu** a priori

structure `for` ou `do` avec compteur



nombre d'itérations **inconnu** a priori

structure `while`

risque de boucle infinie

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 *(au moins 1 fois)* **} while (**expr. **);**	faire ... tant que	

Boucle **`for`**

— **`expr1`** évaluée une fois avant l'entrée dans la boucle

généralement initialisation d'un compteur

— **`expr2`** : condition d'arrêt évaluée avant chaque itération

— **`expr3`** évaluée à la fin de chaque itération

généralement incrémentation du compteur

5.3.1 Exemples de boucle `for` ou `do`

for.c

```
/* affichage des entiers impairs <= m */
/* mise en oeuvre de la structure "for"  */
for (i = 1; i <= m; i = i + 2)
 {
  printf(" %d \n", i) ;          /* un bloc */
 }
printf("-------\n"); /* en dehors du for ! */
```

do.f90

```
! affichage des entiers impairs inférieurs à m
! structure "do" avec compteur
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
```

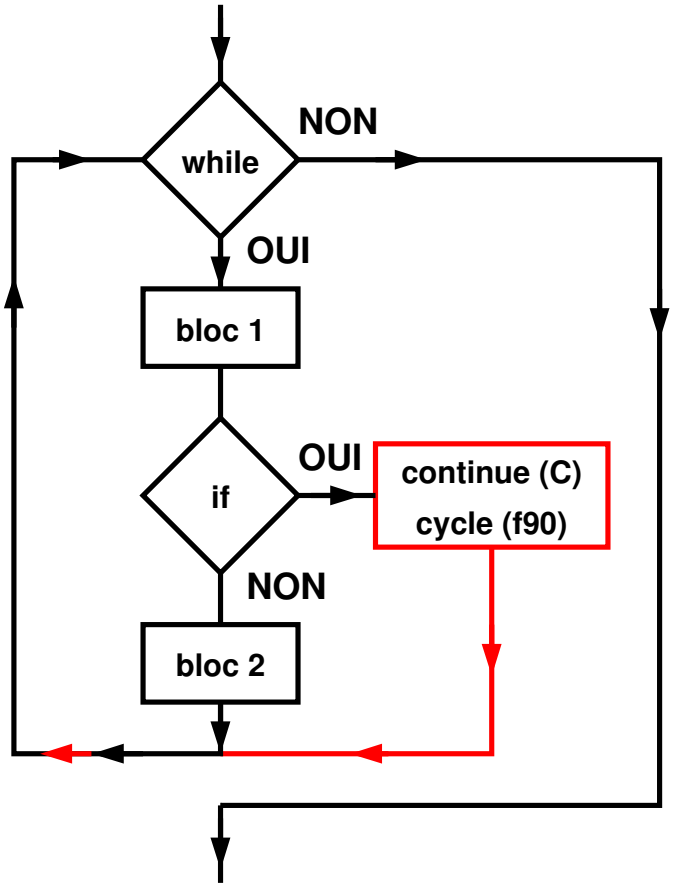
5.4 Branchements ou sauts

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.

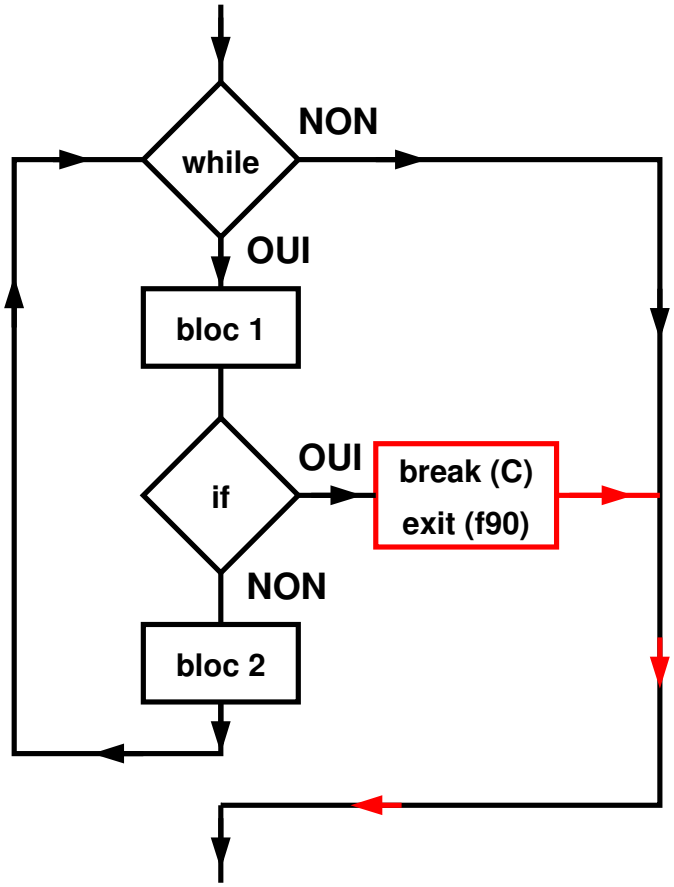
Rebouclage anticipé

`continue` ou `cycle`



Sortie anticipée de boucle

`break` ou `exit`



5.4.1 Exemples de bouclage anticipé `cycle`/`continue`

```c
int i = 0 , m = 11;
while ( i < m ){/* rebouclage anticipé via continue */
  i++ ;
  if ( (i % 2) == 0 ) continue ; /* si i pair */
  printf(" %d \n", i) ;
}
```

```
i = 0 ! recyclage anticipé  via "cycle"
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
```

5.4.2 Exemples de sortie anticipée de boucle via `break`/`exit`

```c
int i = -1 , m = 11;
while ( 1 ) { /* toujours vrai */
  i += 2 ;
  if ( i > m ) break ; /*  sortie de boucle */
  printf(" %d \n", i) ;
}
```

```fortran
i = -1 ! initialisation
DO ! boucle infinie a priori
   i = i + 2
   IF( i> m ) EXIT ! sortie anticipée dès que i > m
   WRITE(*,*) i
END DO
```

6 Introduction aux pointeurs

6.1 Intérêt des pointeurs

— **variables** permettant de désigner successivement différentes variables afin de :

⇒ faire un 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** de variables	pas d'accès aux adresses

— **utilisation commune :** tris sur des données volumineuses, implémentation de structures de données auto-référencées (listes chaînées par ex.)

6.2 Pointeurs et variables : exemple du C

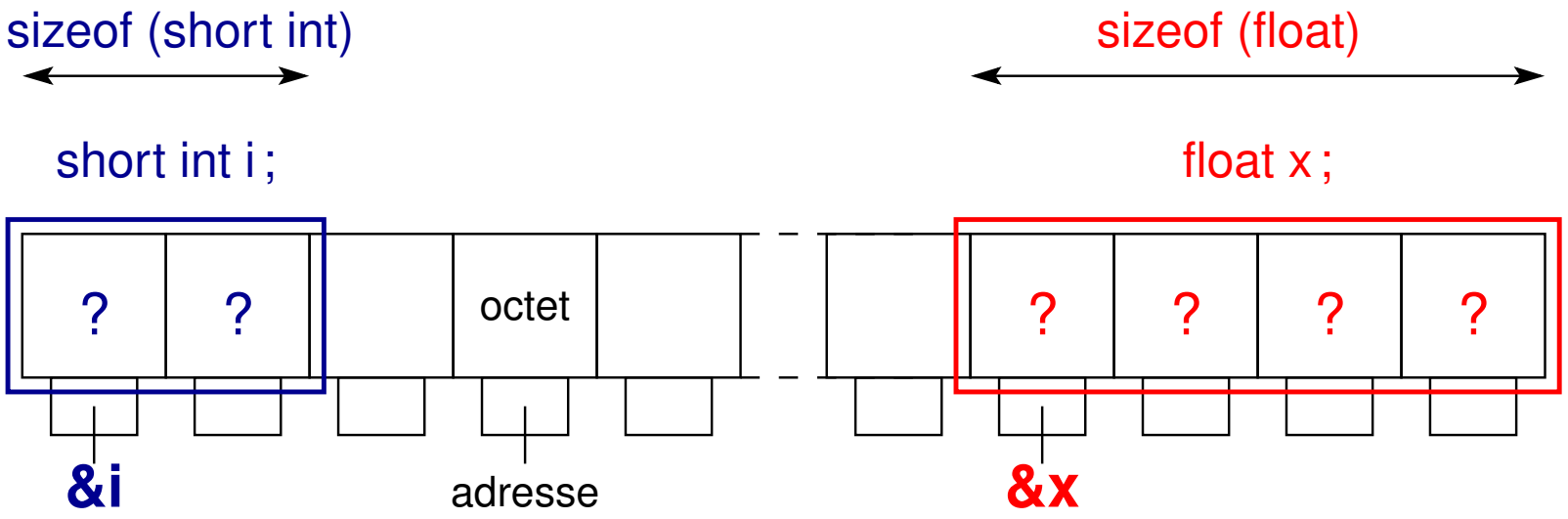
Déclarer une variable d'un type donné =

réserver une zone mémoire (donne une adresse)

dont la taille (`sizeof` en C) et le codage

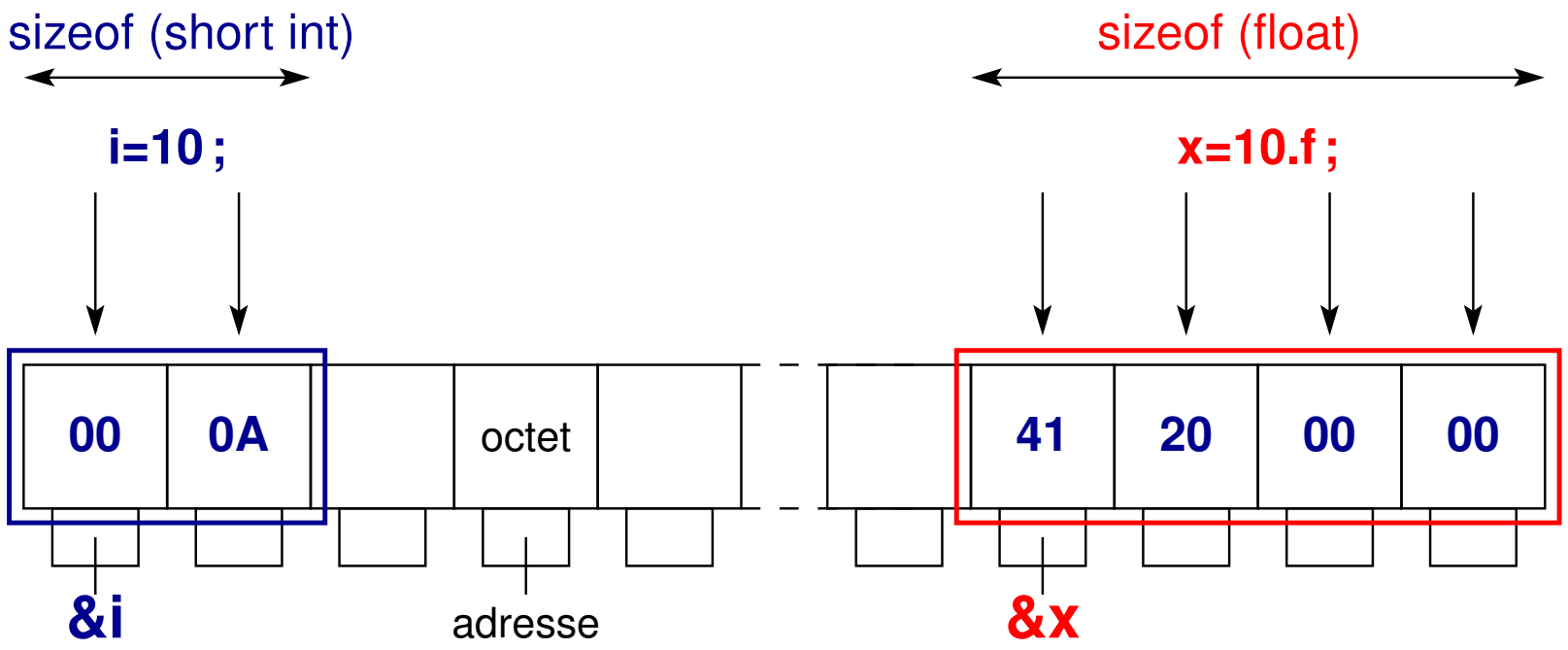
sont fixés par le type

⚠ L'adresse d'une variable peut changer entre deux exécutions



Affecter une valeur à une variable d'un type donné =

écrire la valeur dans les cases réservées selon le codage du type

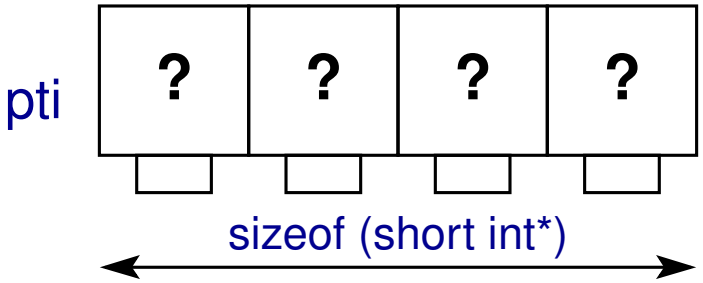


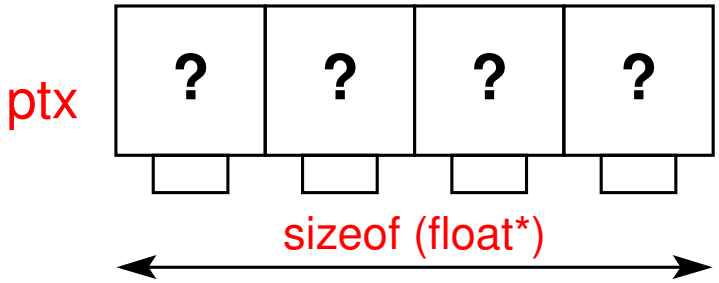
Déclarer une variable pointeur vers un type donné =

réserver une zone mémoire pour stocker **des adresses**

de variables de ce type : **les pointeurs sont typés**

$\Rightarrow$ leur **type** indique la **taille** et le **codage** de la **cible** potentielle





short int *pti ;

pointeur d'entier court

float *ptx ;

pointeur de float

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**

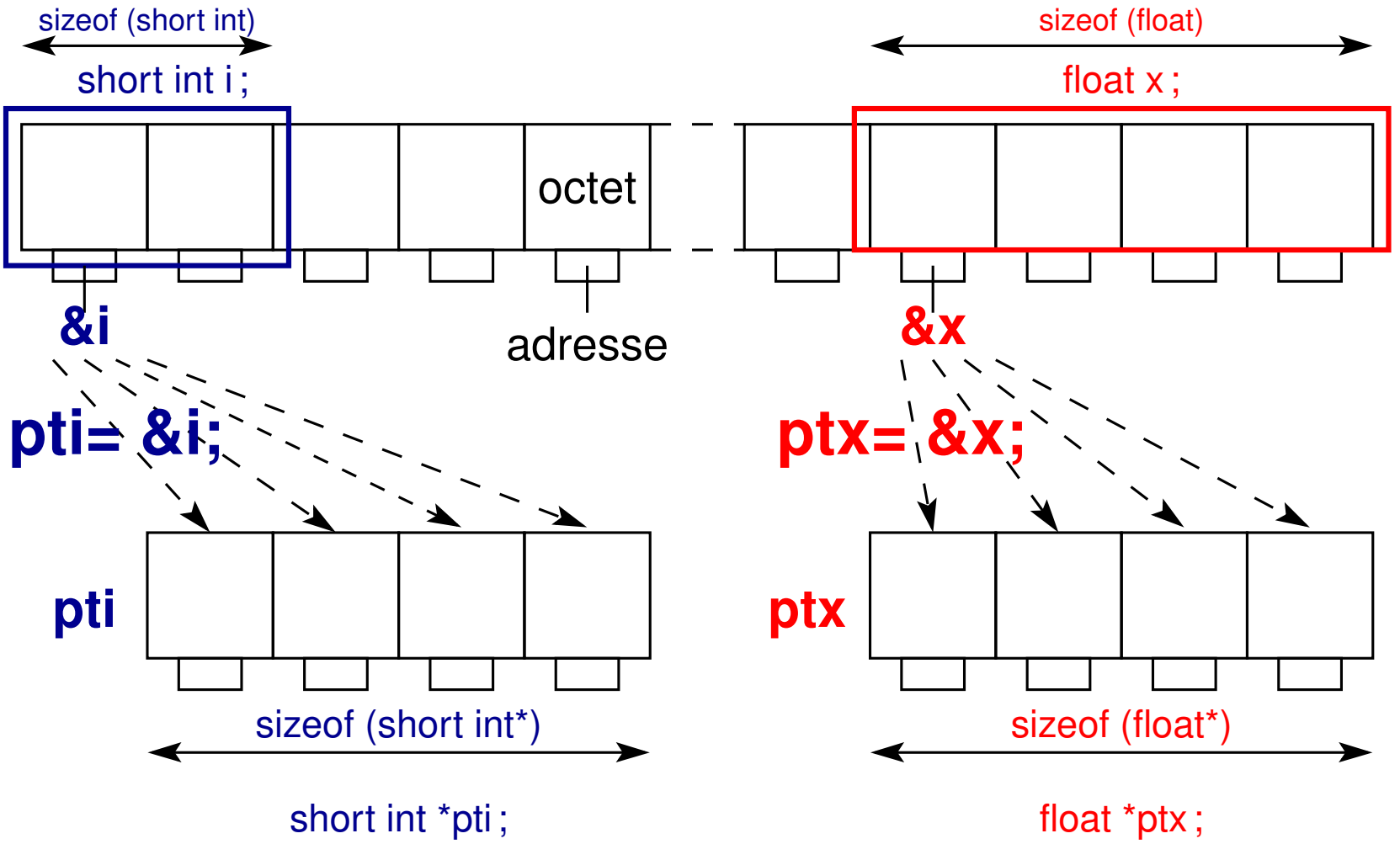
Affecter la **valeur d'un pointeur** `pty` **à un pointeur** de même type `ptx` :

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

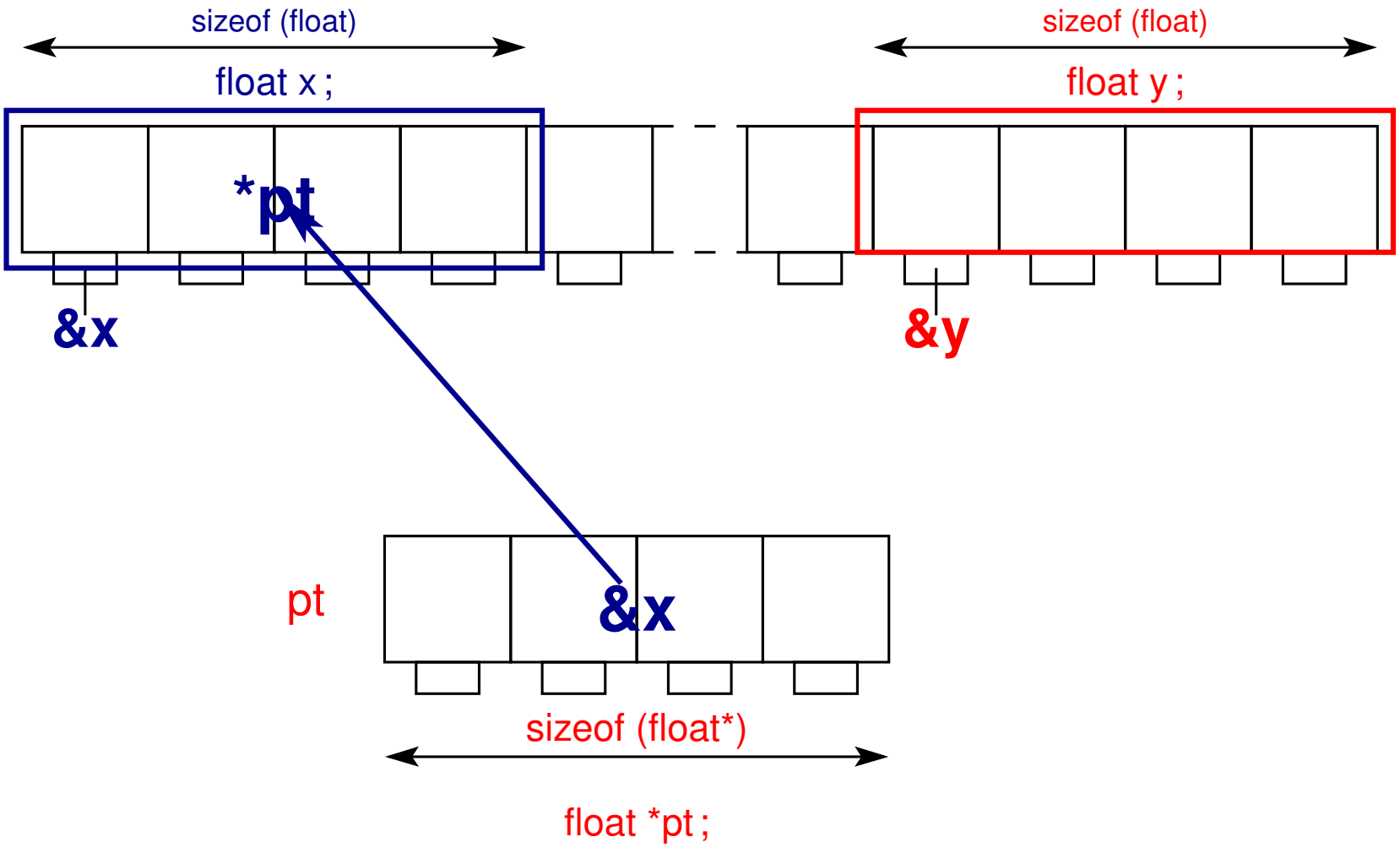
Faire pointer un pointeur vers une variable =

écrire l'adresse de la variable dans les cases réservées pour le pointeur

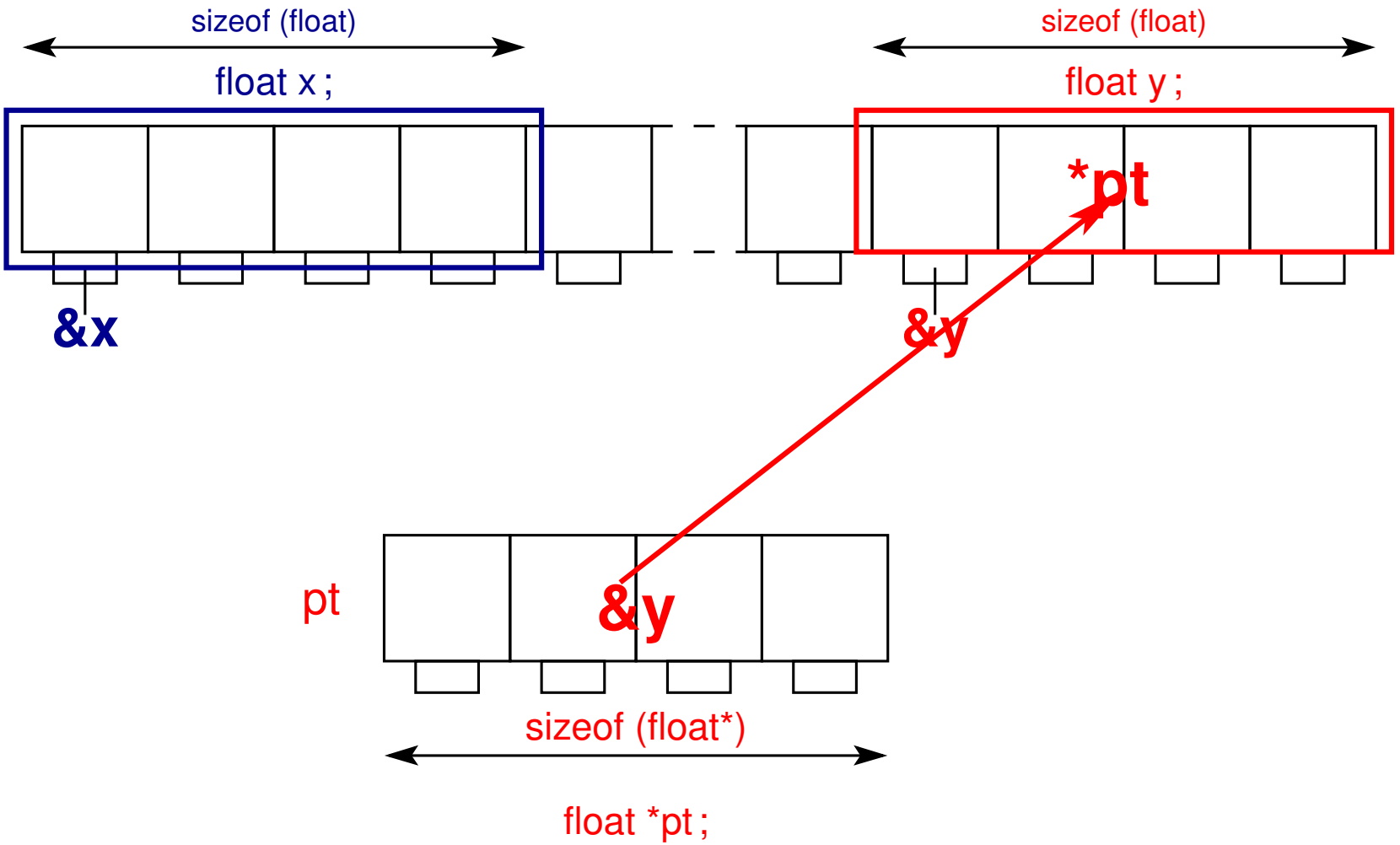
= affecter une valeur à la variable pointeur



Après avoir fait `pt = & x;` **pt pointe vers x**



Après avoir fait `pt = & y;` **pt pointe vers y**



6.2.2 Indirection (opérateur * en C)

L'opérateur **`*`** d'**indirection** permet d'**accéder à 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` contient désormais la valeur `4`

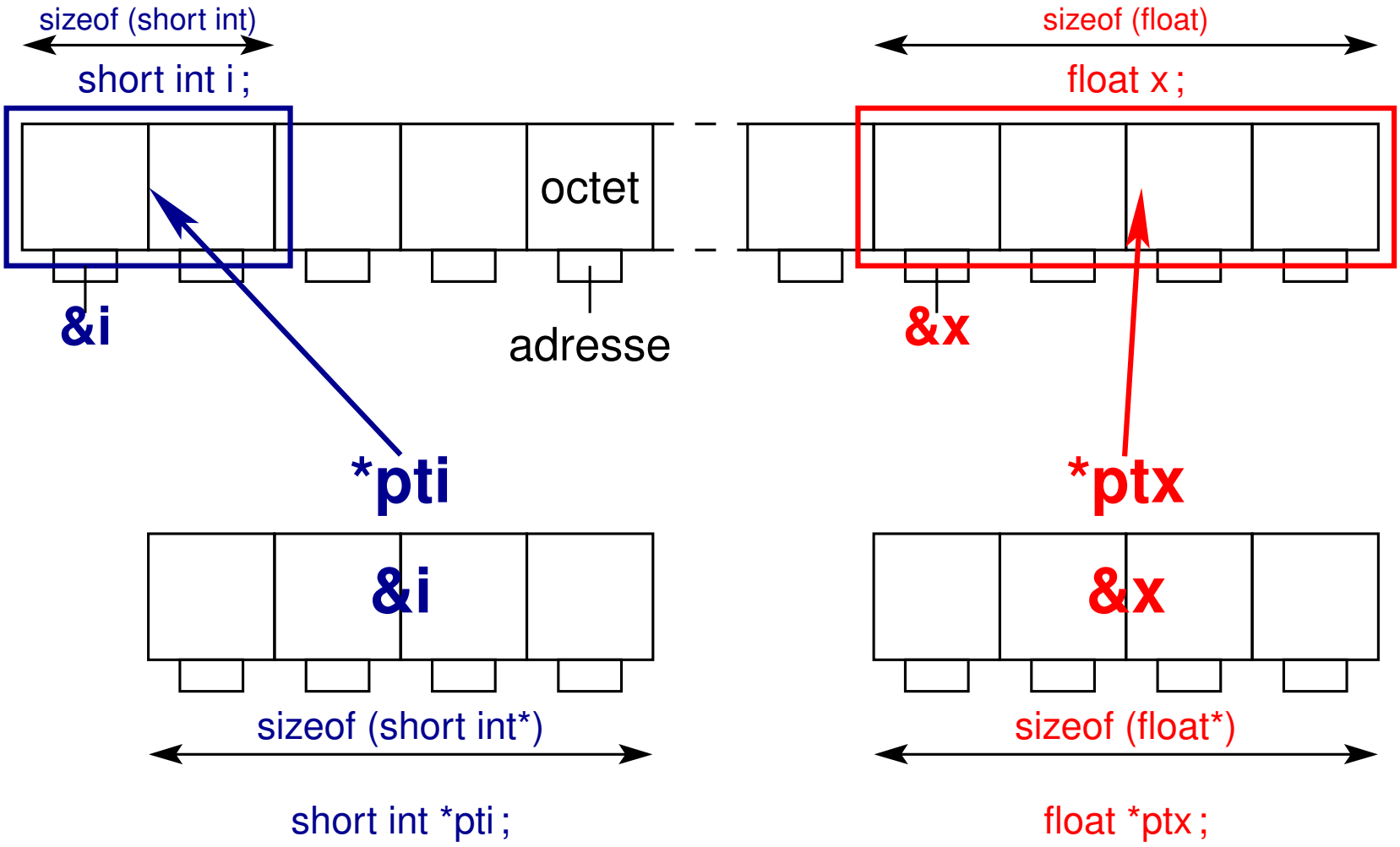
Attention :

— 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 aux variables cibles



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

$\Rightarrow$ 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 potentielle	`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)

6.5.1 Exemple élémentaire de pointeur en C

```c
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) ;
```

6.5.2 Exemple élémentaire de pointeur en fortran

```
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
```

6.6 Initialiser les pointeurs !

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

En C, l'adresse qu'il contient est aléatoire

$\Rightarrow$ le résultat d'une indirection est aussi aléatoire

En fortran, l'état initial d'un pointeur est indéterminé

$\Rightarrow$ initialiser les pointeurs à **`NULL`** ou **`null()`**

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

⚠ En C, **`int * pi = NULL;`**

déclare un pointeur **`pi`** d'entier (`*pi` devra être un entier)

mais affecte **`NULL`** à **`pi`** et non à `*pi`

⚠ **`int * pi, j;`** déclare **`pi`** comme pointeur d'entier mais **`j`** comme entier

erreur-pointeur.c

```
int i;
/* pointeur non initialisé => erreur aléatoire d'accès mémoire */
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\n" " pj=%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 */
```

```
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

$\Rightarrow$ **réutilisation dans plusieurs contextes**

Distinguer :

— Argument **formel** ou **muet** (*dummy*) dans la définition de la procédure

— Argument **effectif** (*actual*) dans l'appelant

Passage des arguments **par copie de valeur en C** mais **par référence en fortran**

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

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

procédures **à effets de bord** (*side effects*)		
fonctions de type **void** **return ;** (sans expression)	entrées/sorties par ex.	**sous-programmes** (*subroutines*) modif. possible des arguments
`fct(x1, x2, ...) ;`	invocation dans l'appelant	**`CALL`** `sub(x1, x2, ...)`

7.1.1 Mode de passage des arguments et conséquences

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 ∃ procédures génériques)

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

7.1.2 Vocation des arguments en fortran

Passage des arguments par référence en fortran $\Rightarrow$ modifiables

$\Rightarrow$ Toujours préciser leur vocation via l'attribut **`INTENT`**

`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 Structure des programmes

7.2.1 Structure d'un programme C

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

Pas d'imbrication des fonctions en C : définition en dehors de toute fonction.

— La définition d'une **fonction** se compose d'un **entête**

et d'un **corps** entre **`{`** et **`}`**

— L'**entête** d'une fonction spécifie le **type** de la valeur de retour, le nom de la fonction et ses paramètres ou arguments avec leur type :

```
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

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

  $\Rightarrow$ définition placée **avant l'appel**
- déclarer en global $\Rightarrow$ visibilité dans tout ce qui suit dans le fichier

  $\Rightarrow$ déclarer en début de fichier et définir plus loin
- 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

7.2.2 Structure générale d'un programme fortran

Un **programme** fortran = un programme principal (**`program`**) plus éventuellement des **procédures** : **fonctions** (**`function`**) et **sous-programmes** (**`subroutine`**)

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) à éviter, sinon fournir l'interface
3. intégrées dans des **modules** externes (introduites par **`contains`**)
   $\Rightarrow$ l'interface est mémorisée dans un fichier de module lors de la compilation
   $\Rightarrow$ **`use nom_module`** permet à l'appelant de connaître l'interface en relisant le fichier de module **`nom_module.mod`**
   $\Rightarrow$ **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`)

7.3 Exemples de fonctions renvoyant une valeur

Dans la **définition** de la fonction **`som`**, **`p`** est un **argument muet (=formel)**

de même que la variable locale **`i`** dans la boucle

```c
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   */
}
```

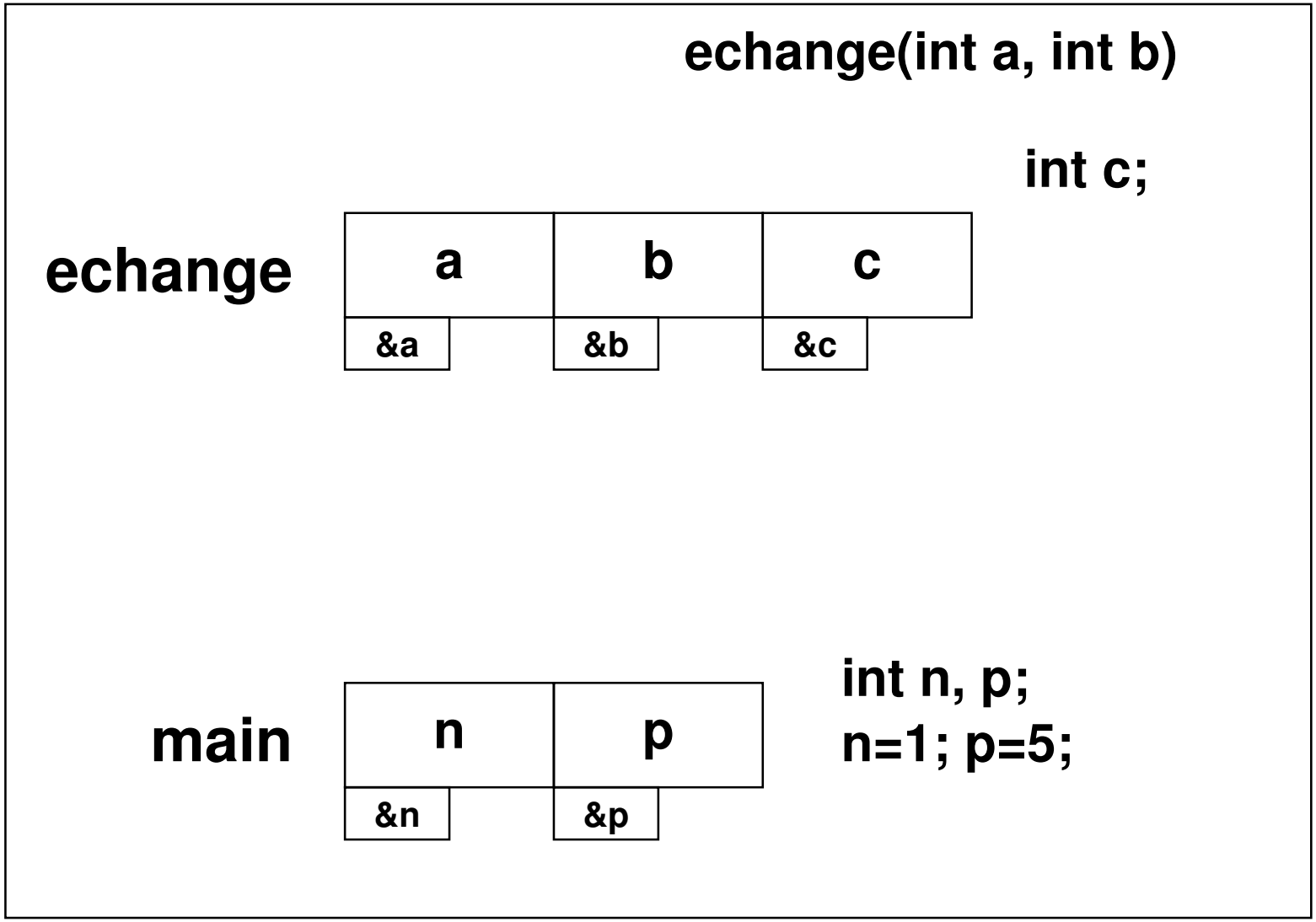
```fortran
INTEGER FUNCTION som(p)
  INTEGER, INTENT(in) :: p
  ! somme des p premiers entiers
  INTEGER :: i, s ! var. locales
  s = 0
  DO i = 0, p
      s = s + i
  END DO
  som = s ! valeur rendue
END FUNCTION som
```

```c
#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) ;
}
```

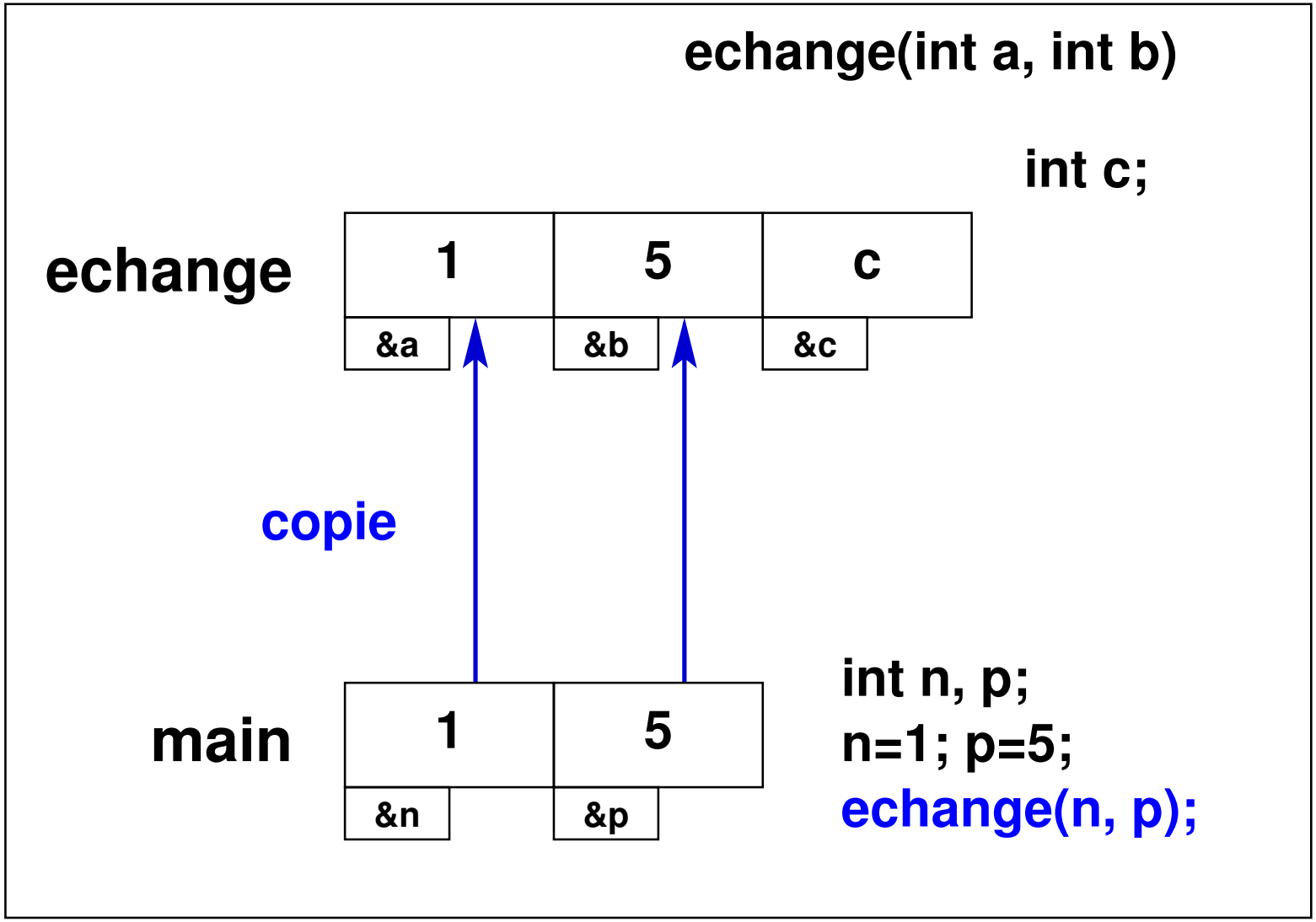
```fortran
PROGRAM ppal
 IMPLICIT NONE
 INTERFACE ! début d'interface
    FUNCTION som(p)
    ! déclaration de la fonction
    INTEGER :: som ! résultat
    INTEGER , INTENT(in) :: p
    END FUNCTION som
    ! fin déclaration de la fct
 END INTERFACE ! fin d'interface
 INTEGER :: s  ! local
 WRITE(*,*)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

7.4 Exemples de procédures

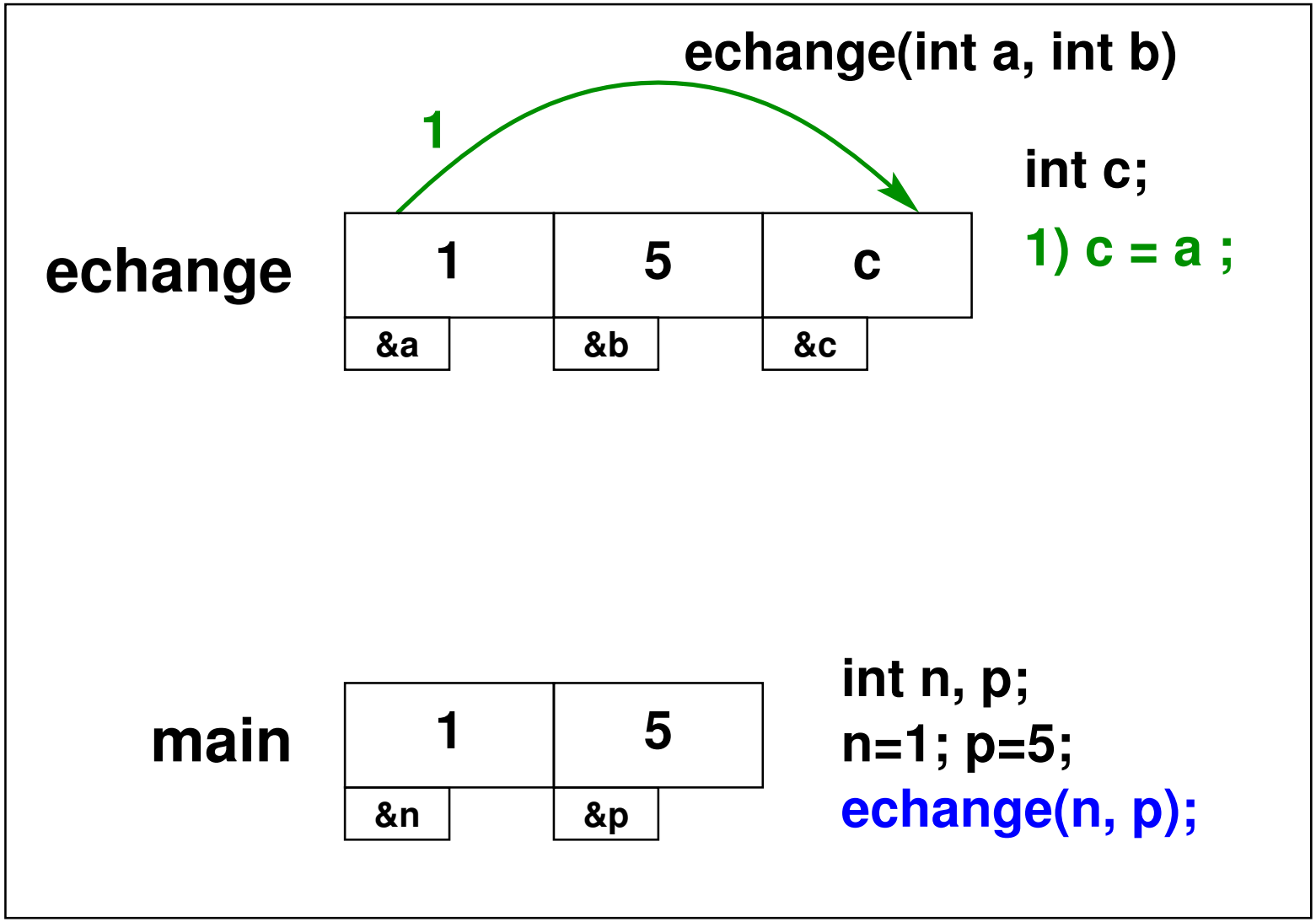
7.4.1 Faux échange en C sans pointeurs



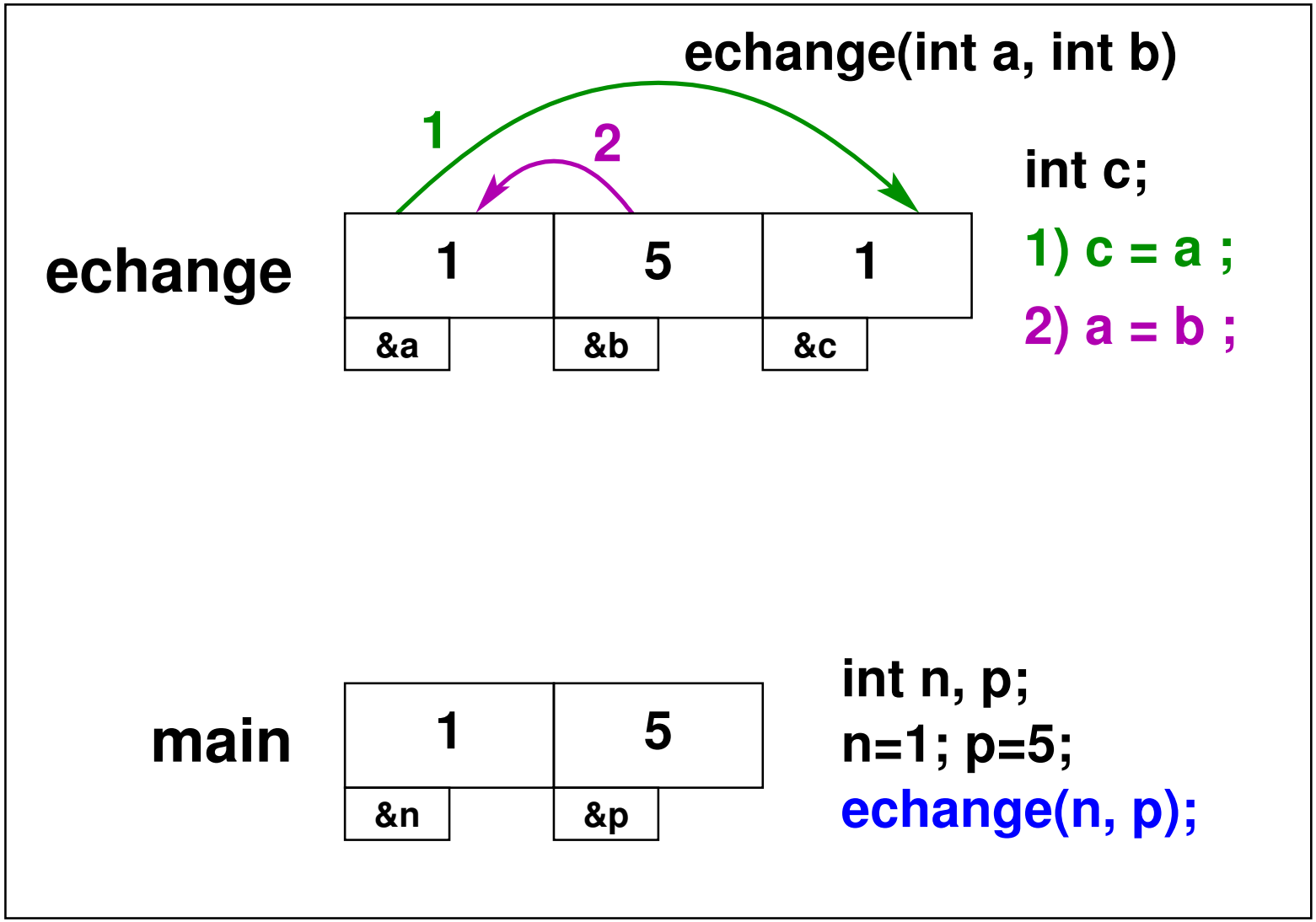
Passage de n et p par copie



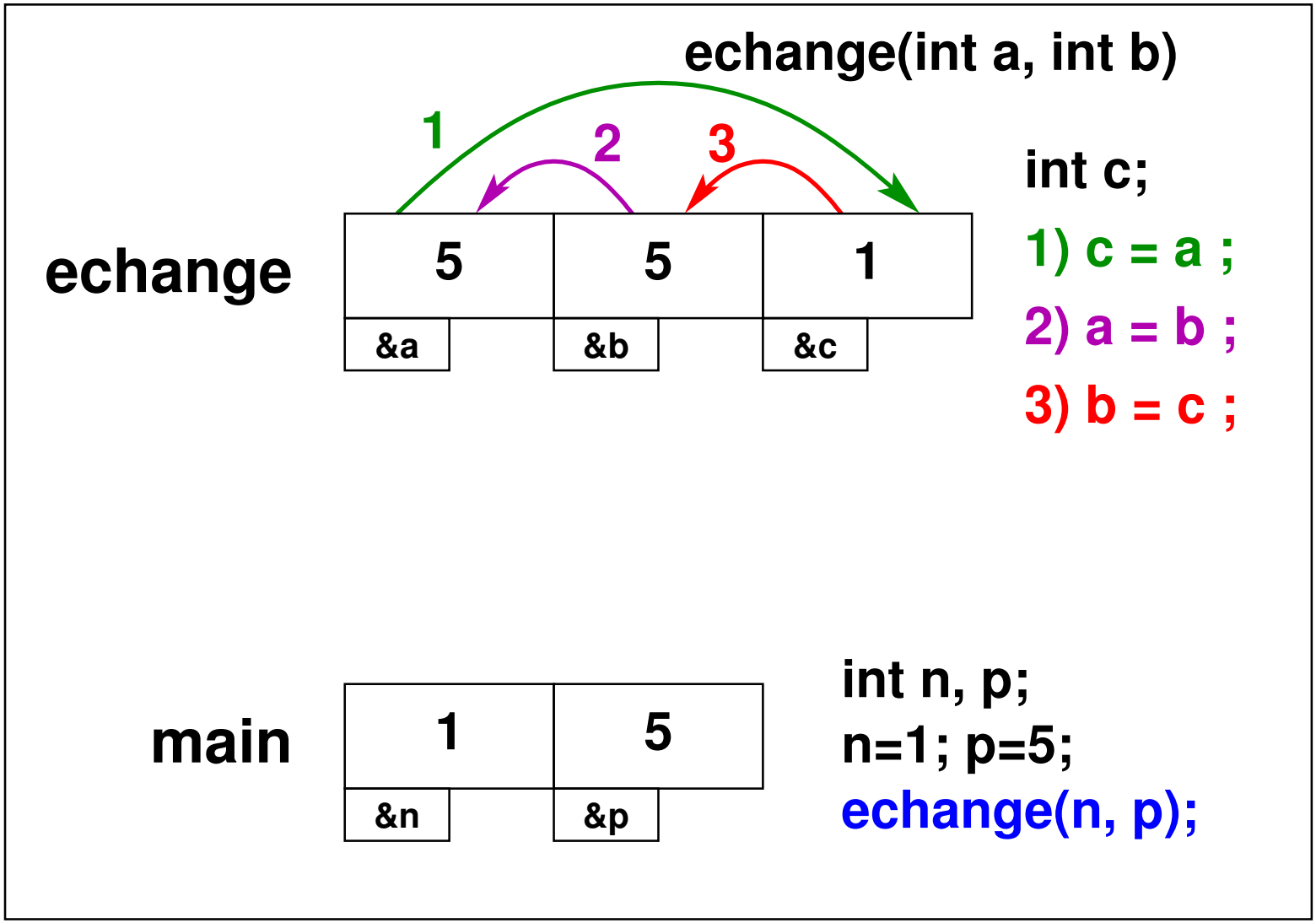
Étape 1 : Stockage de la valeur de `n` dans `c`



Étape 2 : la valeur de b remplace a

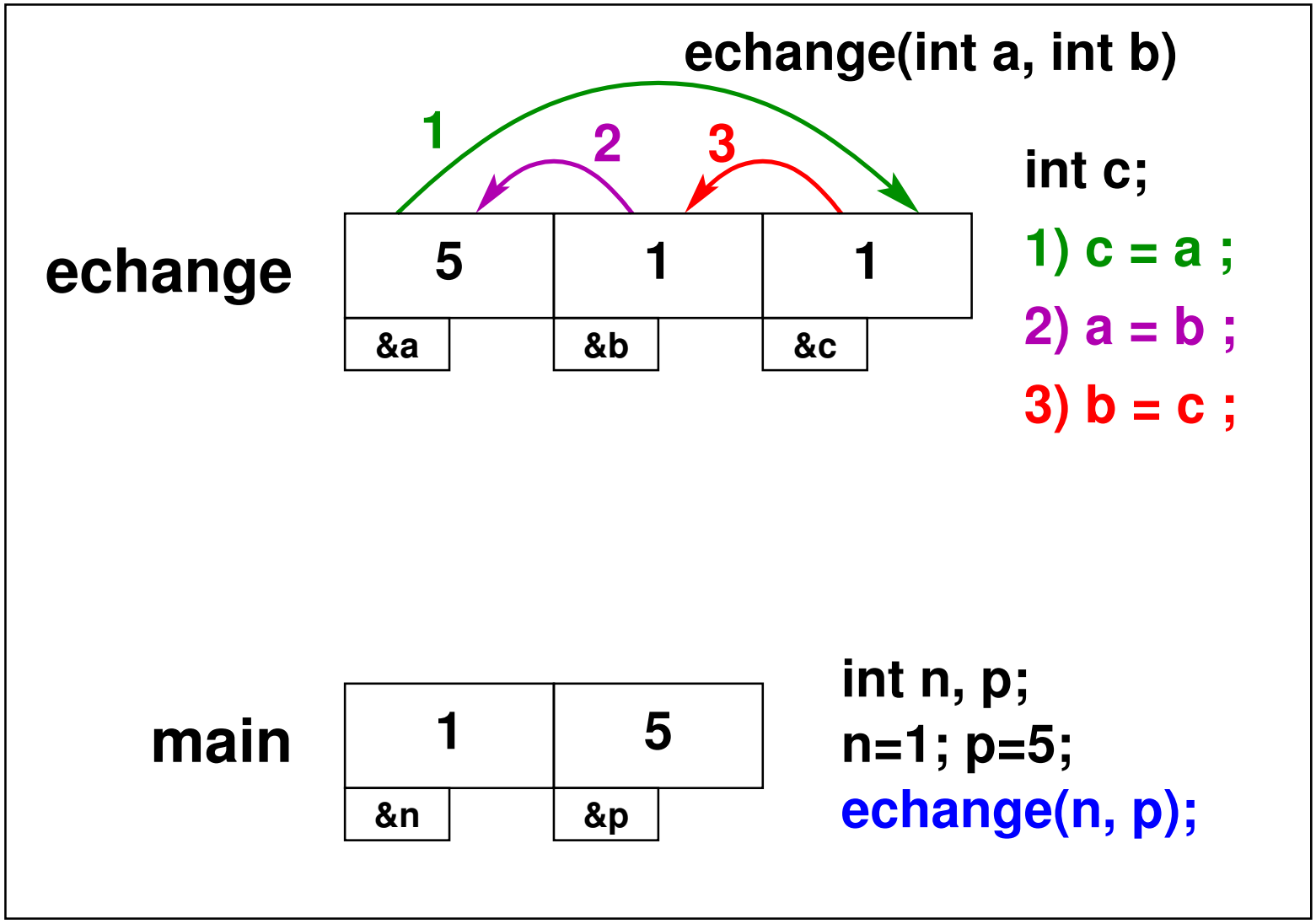


Étape 3 : la valeur de c remplace



b

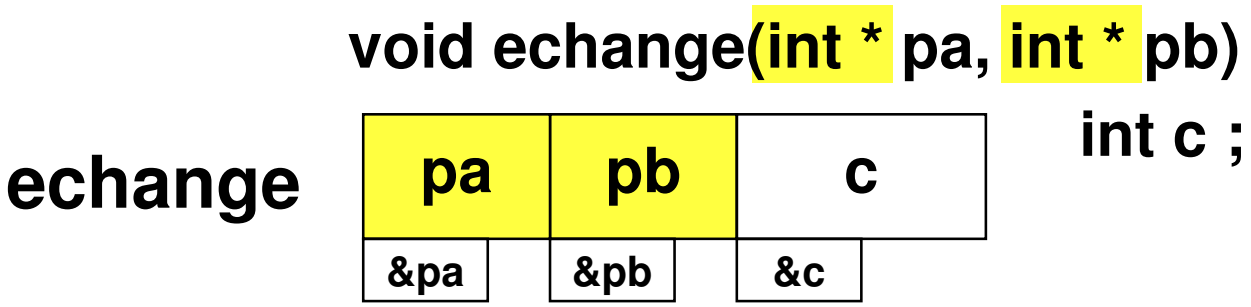
Résumé : permutation entre `a` et `b` mais `n` et `p` inchangés

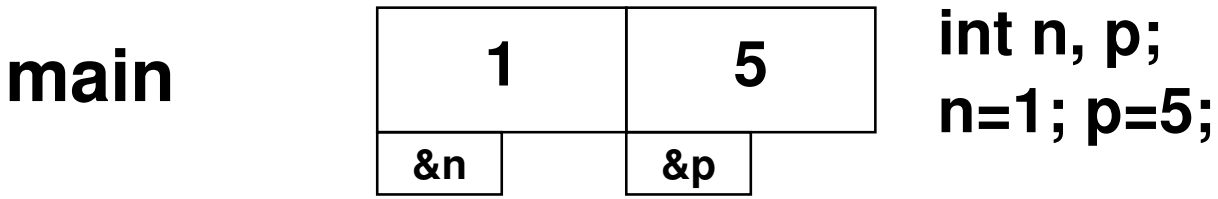


7.4.2 Vrai échange avec pointeurs en C

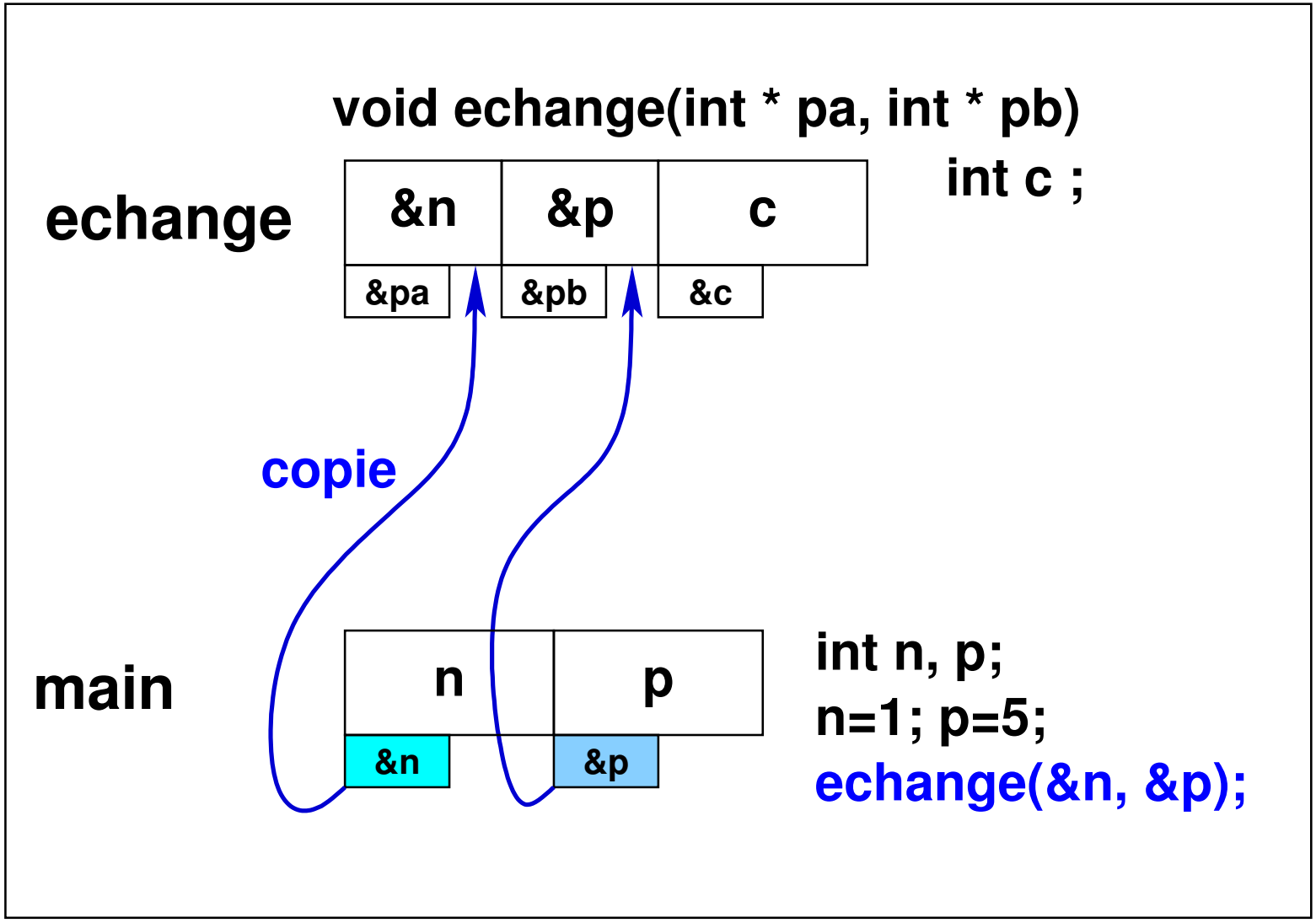
Pour accéder aux variables de l'appelant, passer les copies de leurs adresses

$\Rightarrow$ les stocker dans des variables **pointeurs** vers le type de la cible

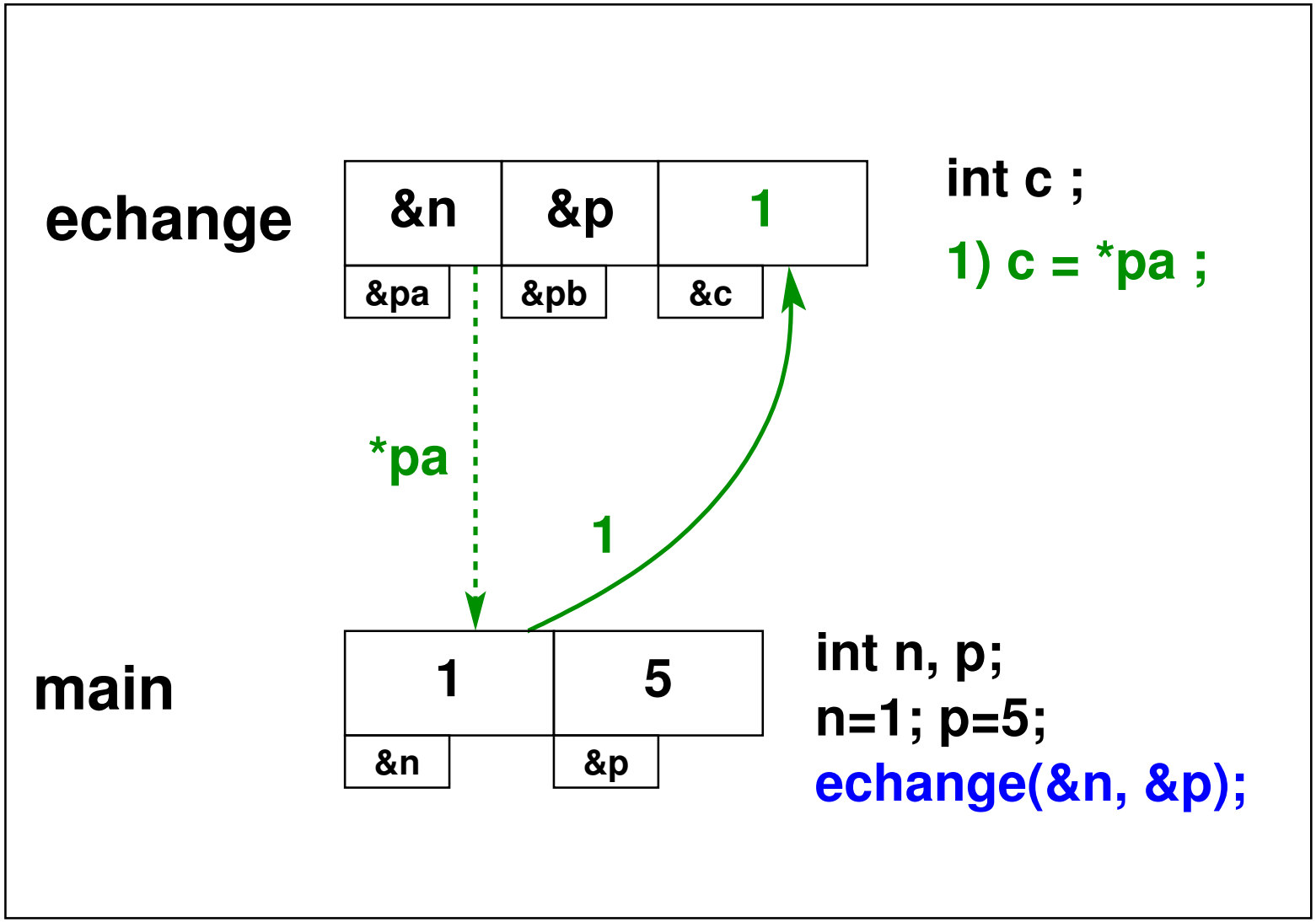




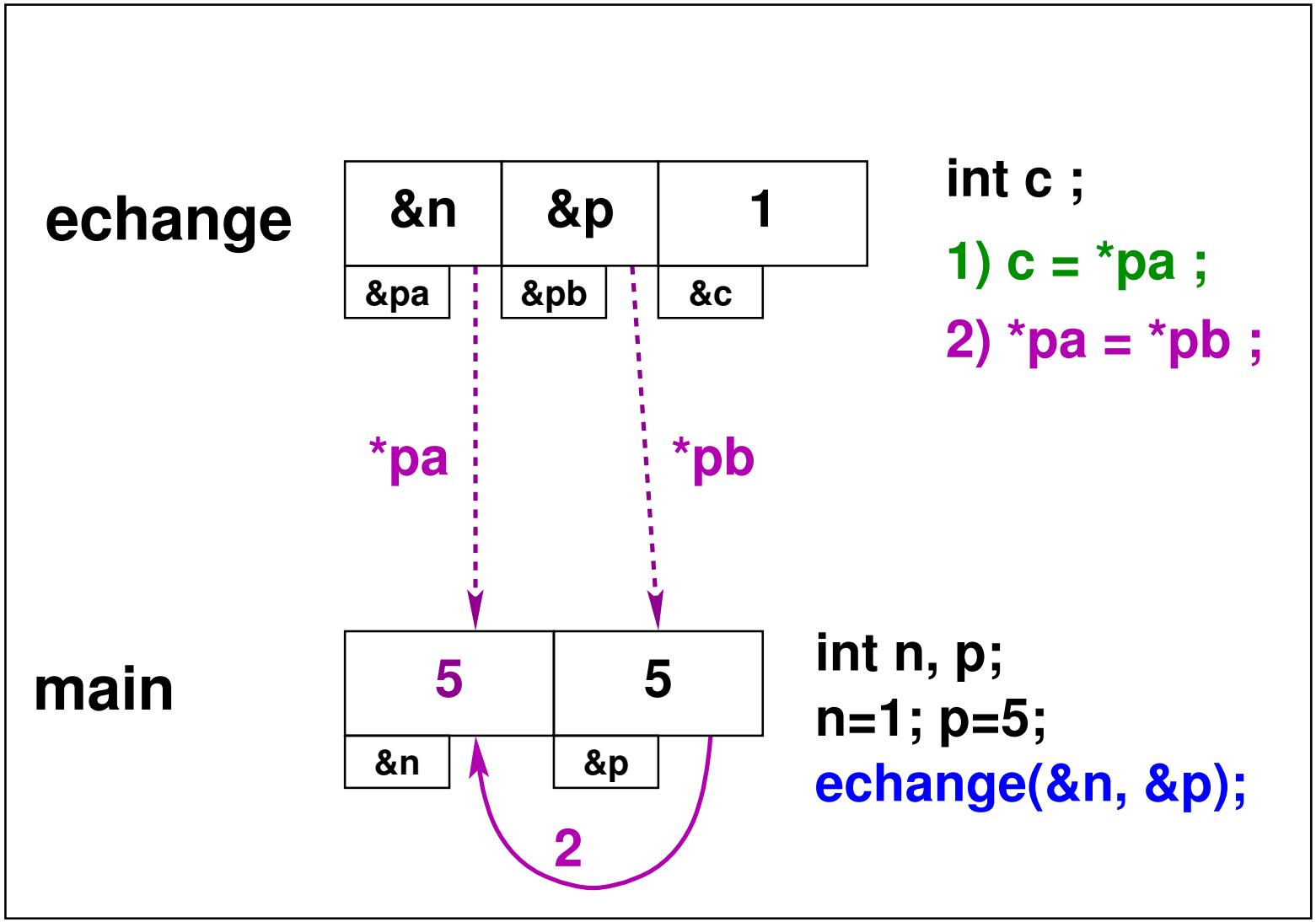
Passage des adresses de n et p par copie : *pa et *pb désignent n et p



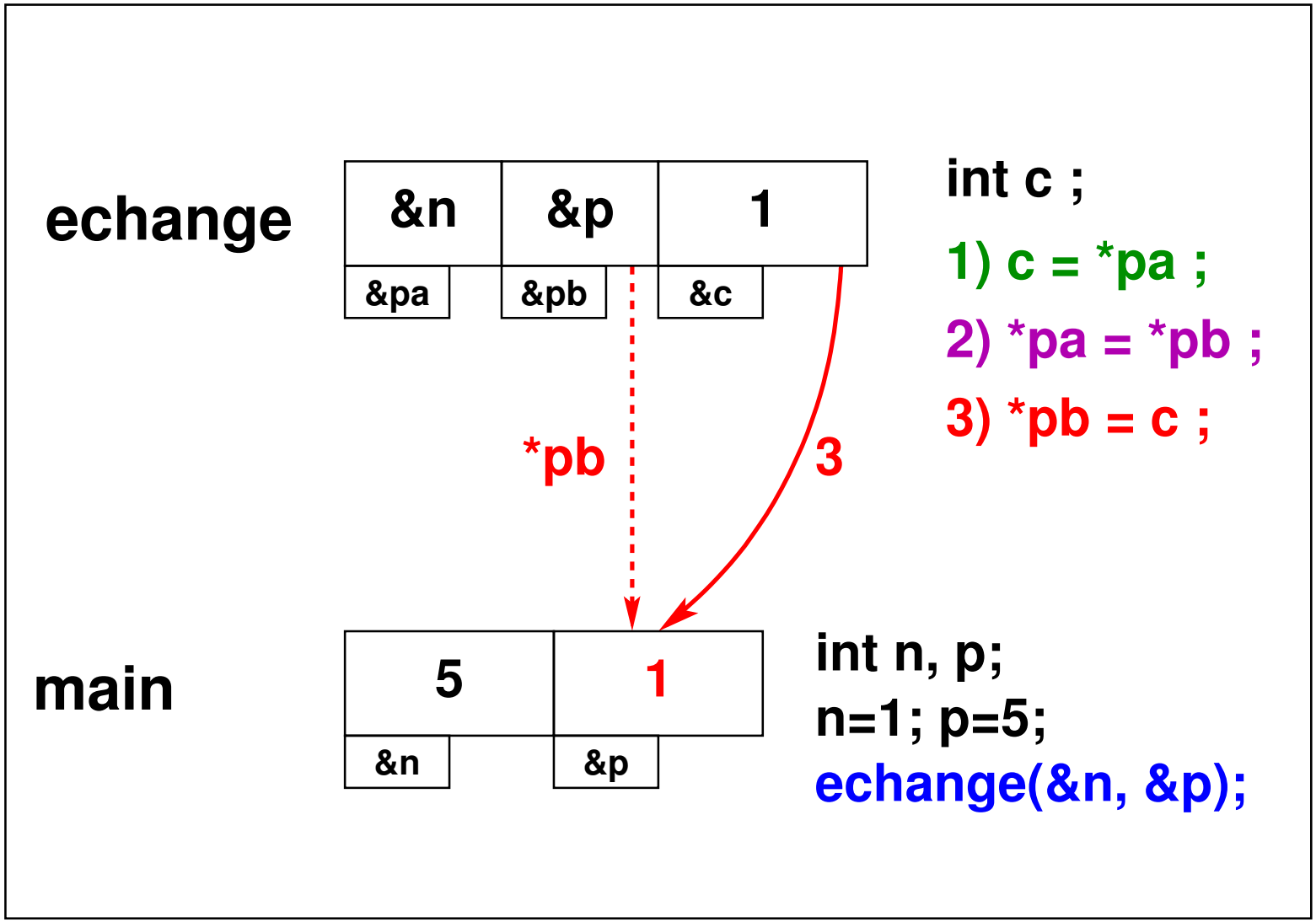
Étape 1 : Stockage de la valeur de `n` dans `c`



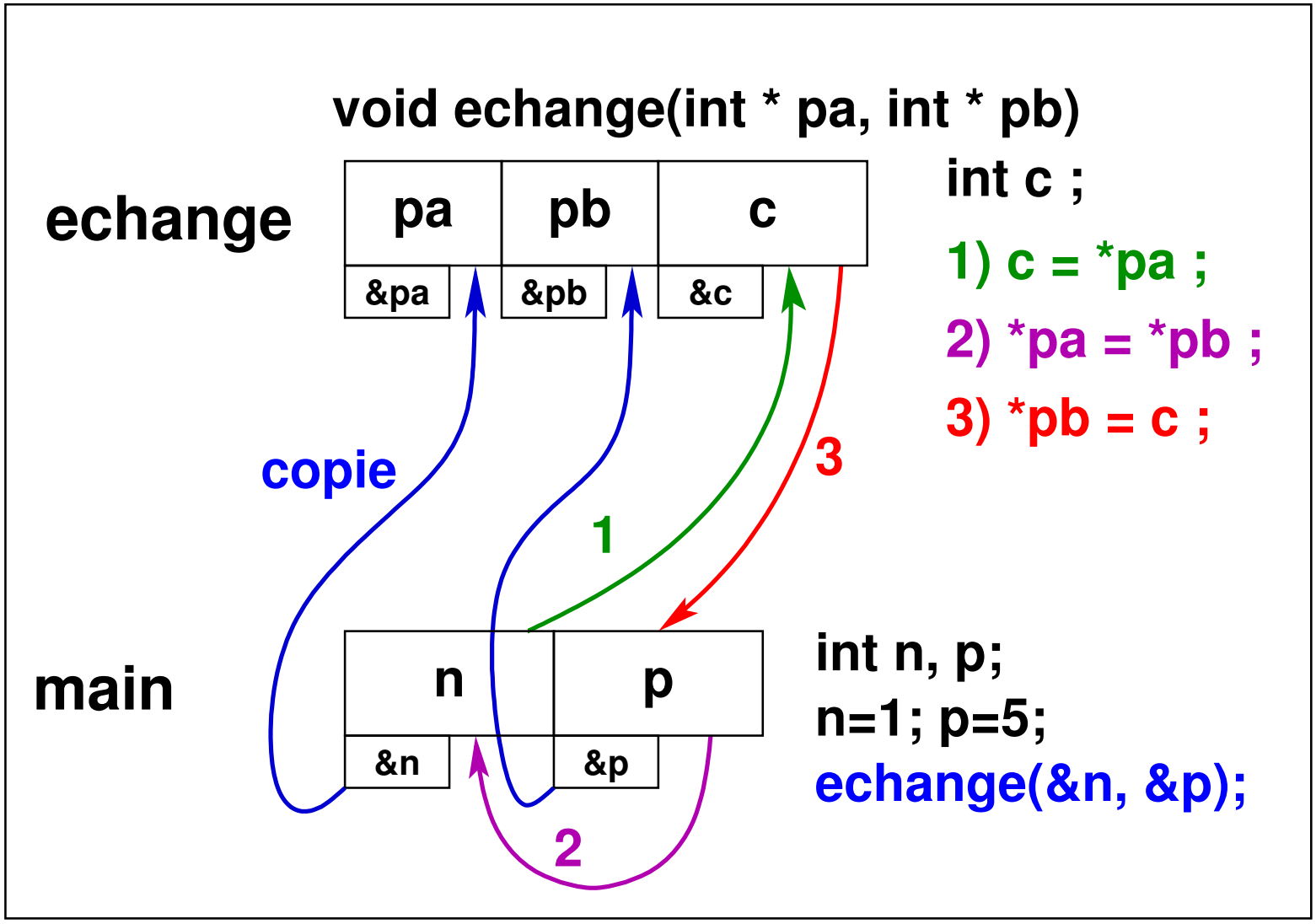
Étape 2 : la valeur de p remplace n



Étape 3 : la valeur de c remplace p



Résumé des opérations : permutation entre `n` et `p`



```c
/* Fonctions : passage des adresses (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 int */
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;
}
```

```c
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(EXIT_SUCCESS) ;
}
```

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
```

7.4.3 Procédure de module en fortran 90

```
! Sous-programme avec arguments ! fct3c.f90
! procédure de module + USE => contrôle interprocédural
MODULE m_sp
  CONTAINS
  SUBROUTINE sp(n, p)
    IMPLICIT NONE
    INTEGER, INTENT(in)  :: n ! arg d'entrée non modifiable
    INTEGER, INTENT(out) :: p ! arg 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 ! donne au compilateur accès 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 dès la compilat.
  WRITE(*,*) "q =", q, "après appel du sous-programme"
  STOP
END PROGRAM principal
```

7.5 Durée de vie et portée des variables

Durée de vie des variables

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

$\Rightarrow$ 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

$\Rightarrow$ pas de mémorisation entre deux appels

<table>
<tr><th>Langage C</th><th>Fortran</th></tr>
<tr><td colspan="2">variables externes ou globales (⇒ permanentes)</td></tr>
<tr><td>à l'extérieur de toute fonction,
visibles dans les fonctions qui
suivent dans le fichier</td><td>déclarées dans un module avant
CONTAINS et accessibles via USE
(COMMON en fortran 77)</td></tr>
<tr><td colspan="2">variables internes ou locales (⇒ temporaires par défaut)
⇒ portée (scope) réduite</td></tr>
<tr><td>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</td><td>à une procédure, à un module
visibles dans la procédure et les sous-procédures internes
fortran 2008 : notion de bloc</td></tr>
<tr><td colspan="2">redéclaration locale d'une variable dont la portée assurait la visibilité
⚠ ⇒ masquage par la variable locale</td></tr>
</table>

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 $\not\Rightarrow$ 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 mod1,only:var1,fct2`**

7.5.1 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 décla
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(EXIT_SUCCESS);
}
```

Décompte de 1 si p déclaré avant q

Compte de 1 si q déclaré avant p

Pas de passage explicite de variable

$\Rightarrow$ Comportement aléatoire

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

```c
#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(EXIT_SUCCESS);
}
```

```
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 inconnu dans le programme principal
    CALL avance   ! appel du sous programme sans argument
  END DO
END PROGRAM t_compte
```

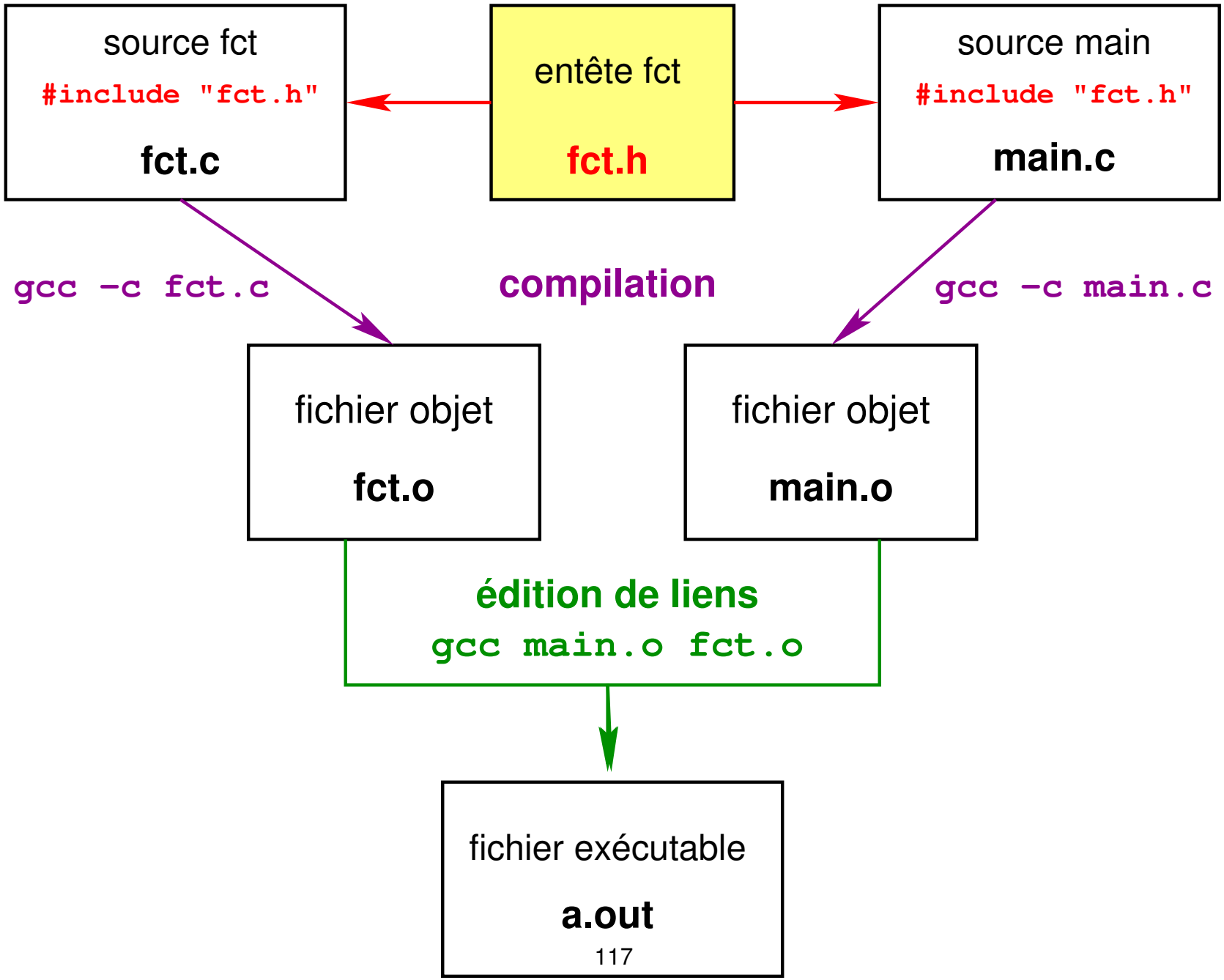
7.5.3 Exemples d'usage de variable globale

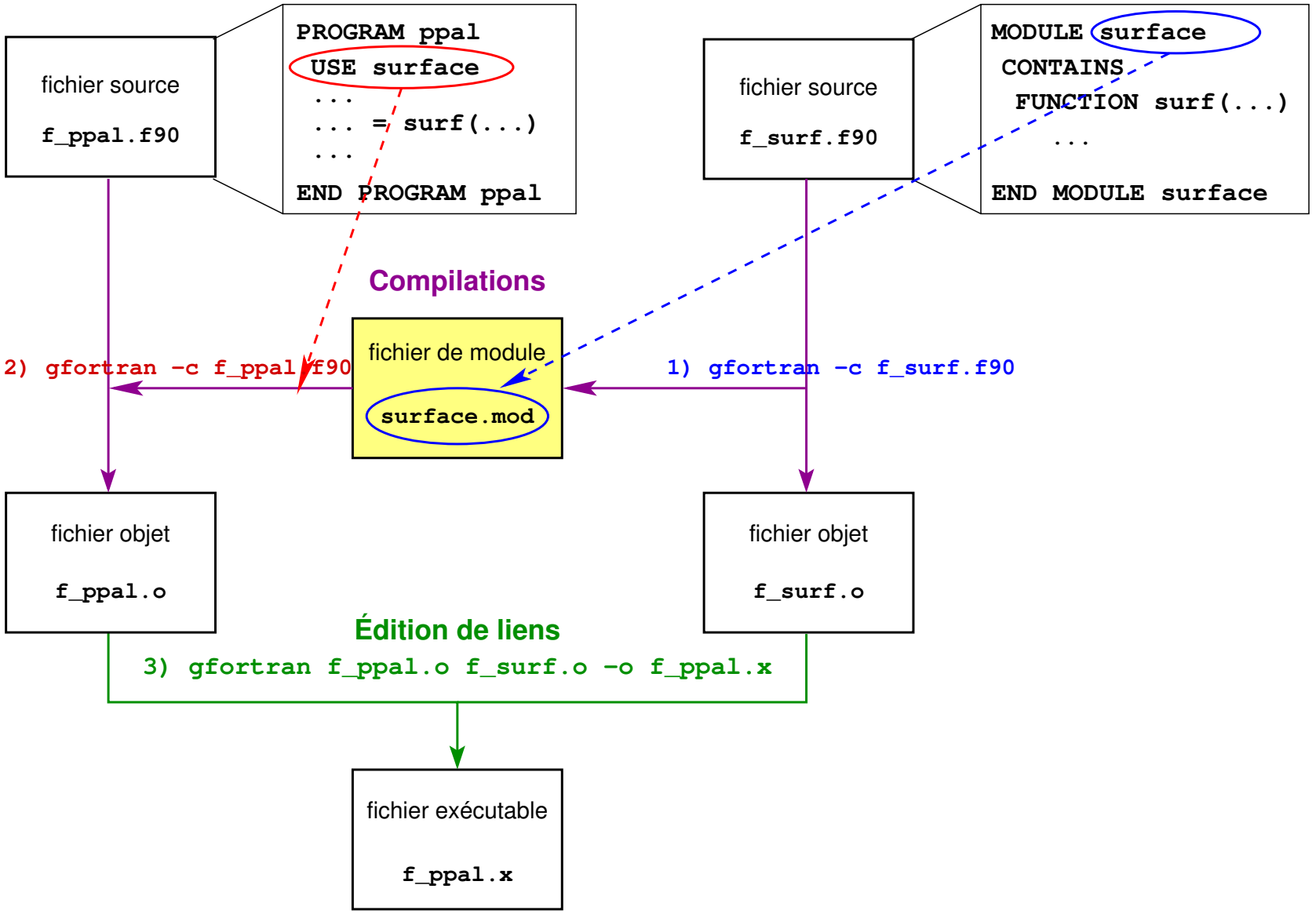
```c
#include<stdio.h> /* fichier compte-globale2.c */
#include<stdlib.h>
/* n= globale pour communiquer entre la fct avance et le main */
int n;  /* variable globale déclarée avant avance */
void avance(void) ; /* déclaration de avance */
void avance(void) { /* définition de avance */
  n++;                    /* 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(EXIT_SUCCCESS);
}
```

```
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
```

7.6 Visibilité des interfaces et compilation séparée

Langage C	Fortran 90
	procédures **internes** introduites par **CONTAINS** $\Rightarrow$ 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**


source fct
#include "fct.h"
fct.c
entête fct
fct.h
source main
#include "fct.h"
main.c
gcc -c fct.c
compilation
gcc -c main.c
fichier objet
fct.o
fichier objet
main.o
édition de liens
gcc main.o fct.o
fichier exécutable
a.out


fichier source
f_ppal.f90
PROGRAM ppal
USE surface
...
... = surf(...)
...
END PROGRAM ppal
fichier source
f_surf.f90
MODULE surface
CONTAINS
FUNCTION surf(...)
...
END MODULE surface
Compilations
fichier de module
surface.mod
2) gfortran -c f_ppal.f90
1) gfortran -c f_surf.f90
fichier objet
f_ppal.o
fichier objet
f_surf.o
Édition de liens
3) gfortran f_ppal.o f_surf.o -o f_ppal.x
fichier exécutable
f_ppal.x

7.7 Compléments sur les procédures

Langage C	Fortran 90
Récursivité des procédures	
récursivité autorisée pour les fonctions par défaut en C	déclarer l'attribut **`RECURSIVE`** et la variable résultat par **`RESULT`**`(var)` (fcts)
nb d'arguments variable (ex. `printf`) 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** `void *` pour manipuler plusieurs types de données (tri avec `qsort`)	procédures **génériques** : même appel pour différents types (suppose la présence de versions spécifiques pour chaque type) les fcts mathématiques intrinsèques sont génériques création de fonctions génériques utilisateur

7.7.1 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;`

Fonction factorielle récursive en fortran

```
!                                        fichier t_fct_recurs.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_recurs+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_recurs+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
```

7.7.2 Les fonctions mathématiques

Langage C : C89, C99	fortran
⚠ `#include <math.h>` pour le compilateur C89	bibliothèque intrinsèque
⚠ **`-lm`** pour l'éditeur de lien (`libm.a`)	ni **`use`**, ni option **`-l`**
Généricité des fonctions mathématiques	
`#include <tgmath.h>` pour le compilateur C99	génériques par défaut
appelle versions **`float`** et **`long double`** selon arg.	
ex : **`sin`** (générique) appelle `sin`**`f`**, `sin` ou `sin`**`l`**	
`man 3 fct` pour le manuel où **`fct`** est la version générique	

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.

Type des arguments des fonctions et de leurs résultats (notés $\rightsquigarrow$)
dans les tableaux des fonctions suivants

	entiers	réels	complexes
arguments	**`n`**, **`p`**	**`a`**, **`b`**	**`c`**
résultat $\rightsquigarrow$	**`i`**	**`r`**	**`z`**
variantes en C	**`int`**	**`float`**	**`float complex`**
	`long`	**`double`**	**`double complex`**
	`long long`	**`long double`**	**`long double complex`**

fortran	C89	⇝	C99 + tgmath.h	⇝	remarques
⚠ ABS(n)	l/ll/**abs**(n)	**i**	**abs(n)**	**i**	**abs** pour les entiers en C
ABS(a)	**f**abs/f/l(a)	**r**	**fabs(a)**	**r**	$\lvert a\rvert$
ABS(c)	**c**abs/f/l(c)	**r**	**fabs(c)**	**r**	$\lvert c\rvert$
SQRT(a)	**sqrt**/f/l(a)	**r**	**sqrt(c)**	**z**	
a**b	**pow**/f/l(a,b)	**r**	**pow(c,b)**	**z**	opérateur en fortran
EXP(a)	**exp**/f/l(a)	**r**	**exp(c)**	**z**	$\exp(c)$
LOG(a)	**log**/f/l(a)	**r**	**log(c)**	**z**	$\ln(c)$
LOG10(a)	**log10**/f/l(a)	**r**	**log10(a)**	**r**	$\log(a)$
REAL(a)			**creal(c)**	**r**	$\Re(c)$
AIMAG(a)			**cimag(c)**	**r**	$\Im(c)$
CONJ(a)			**conj(c)**	**z**	c^*
CEILING(a)	**ceil**/f/l(a)	**r**	**ceil(a)**	**r**	$\lceil a\rceil$ ⇝flottant en C
FLOOR(a)	**floor**/f/l(a)	**r**	**floor(a)**	**r**	$\lfloor a\rfloor$ ⇝flottant en C
NINT(a)	l/**lrint**/f/l(a)	**i**	l/**lrint(a)**	**i**	⇝entier long en C
MOD(n,p)	n%p	**i**			opérateur en C
MODULO(n,p)	n%p	**i**			opérateur en C
SIGN(a,b)	**copysign**/f/l(a,b)	**r**	**copysign(a,b)**	**r**	$\lvert a\rvert\times$ signe(b)

fortran	C89	$\rightsquigarrow$	**C99 + tgmath.h**	$\rightsquigarrow$	remarques
`COS(a)`	**cos**`/f/l(a)`	**r**	**cos(c)**	**z**	$\cos(a)$
`COSH(a)`	**cosh**`/f/l(a)`	**r**	**cosh(c)**	**z**	$\cosh(a)$
`ACOS(a)`	**acos**`/f/l(a)`	**r**	**acos(c)**	**z**	$\arccos(a)$
`SIN(a)`	**sin**`/f/l(a)`	**r**	**sin(c)**	**z**	$\sin(a)$
`SINH(a)`	**sinh**`/f/l(a)`	**r**	**sinh(c)**	**z**	$\sinh(a)$
`ASIN(a)`	**asin**`/f/l(a)`	**r**	**asin(c)**	**z**	$\arcsin(a)$
`TAN(a)`	**tan**`/f/l(a)`	**r**	**tan(c)**	**z**	$\tan(a)$
`TANH(a)`	**tanh**`/f/l(a)`	**r**	**tanh(c)**	**z**	$\tanh(a)$
`ATAN(a)`	**atan**`/f/l(a)`	**r**	**atan(c)**	**z**	$\Re \in [-\pi/2, +\pi/2]$
`ATAN2(a,b)`	**atan2**`/f/l(a,b)`	**r**	**atan2(a,b)**	**r**	$\in [-\pi, +\pi]$

fortran **2008**	C89 $\rightsquigarrow$**r**	**C99 + tgmath.h** $\rightsquigarrow$	remarques
ACOSH(a)	**acosh**/f/l(a)	**acosh(c)** **z**	argch(c)
ASINH(a)	**asinh**/f/l(a)	**asinh(c)** **z**	argsh(c)
ATANH(a)	**atanh**/f/l(a)	**atanh(c)** **z**	argth(c)
BESSEL_J0(a)	**j0**/f/l(a)	⚠ les fonctions de Bessel	$J_0(a)$
BESSEL_J1(a)	**j1**/f/l(a)	j0, j1, jn, y0, y1 et	$J_1(a)$
BESSEL_JN(n,a)	**jn**/f/l(n,a)	yn ne sont pas standard	$J_n(n,a)$
BESSEL_Y0(a)	**y0**/f/l(a)	en C, y compris dans les	$Y_0(a)$
BESSEL_Y1(a)	**y1**/f/l(a)	normes C99 ou C11	$Y_1(a)$
BESSEL_YN(n,a)	**yn**/f/l(n,a)		$Y_n(n,a)$
ERF(a)	**erf**/f/l(a)	**erf(a)** **r**	$\frac{2}{\sqrt{\pi}}\int_0^a e^{-t^2}\,\mathrm{d}t$
ERFC(a)	**erfc**/f/l(a)	**erfc(a)** **r**	$\frac{2}{\sqrt{\pi}}\int_a^\infty e^{-t^2}\,\mathrm{d}t$
GAMMA(a)	**tgamma**/f/l(a)	**tgamma(a)** **r**	$\Gamma(a)$
LOG_GAMMA(a)	**l**gamma/f/l(a)	**l**gamma(a) **r**	$\ln(\lvert\Gamma(a)\rvert)$
HYPOT(a,b)	**hypot**/f/l(a,b)	**hypot(a,b)** **r**	$\sqrt{a^2+b^2}$
NORM2(array)			norme euclidienne

8 Tableaux

8.1 Définition et usage

Un **tableau** est un ensemble «**rectangulaire**» d'éléments **de même type**,

stockés de façon contigüe en mémoire en C (pas imposé en fortran)

et repérés au moyen d'**indices entiers**.

L'ensemble de ces éléments 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 taille fixe

Langage C	fortran 90
Déclaration	
`int tab1[3] ;`	`INTEGER, DIMENSION(3) :: tab1`
`int tab2[3][2] ;`	`INTEGER, DIMENSION(3,2) :: tab2`
	`INTEGER :: tab2(3,2), tab1(3)`
Référence à un élément du tableau	
`i = tab1[2] ;`	`i = tab1(3)`
`i = tab2[2][0];`	`i = tab2(3,1)`
⚠ **opérateur séquentiel « , »** ⇒ `[2,0]` **interprété comme** `[0]` **! mais avertissement compilateur**	

Langage C	fortran 90
Indexation différente	
⚠ **commence à 0 ⇒ termine à N−1** possibilité de décaler l'indexation avec un pointeur	**commence à 1 par défaut** **mais choix possible à la déclaration** ex : `DIMENSION(-2:2)` ⇒ de -2 à $+2$
Rangement différent en N dim ⇒ conséquence sur les performances	
indice rapide = qui défile le plus vite = celui des éléments contigus	
⚠ **le plus à droite = le dernier** **tabl. 2D = tableau de tableaux 1D** **⇒ matrices rangées par lignes**	**le plus à gauche = le premier** **⇒ matrices rangées par colonnes**
Constructeurs de tableaux	
`int tab1[3]={1,2,3};`	`INTEGER,DIMENSION(3)::tab1=(/1,2,3/)` `tab1=[1,2,3]` **en fortran 2003**
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 python(numpy), matlab/scilab/octave (mais vecteur ligne = vecteur colonne = tableau 1D) :

- généralisation des opérateurs et des fonctions scalaires « élémentaires »
  aux tableaux **conformants** (même profil) `cos(tab)+ tab`
- promotion des scalaires dans les opérations avec les tableaux `tab =0`
- affectation globale `tab1 = tab2`
- **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

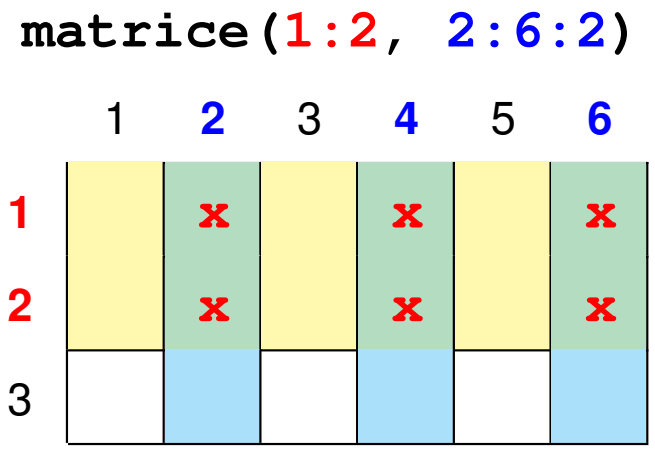
8.3.1 Opérations globales sur les tableaux en fortran

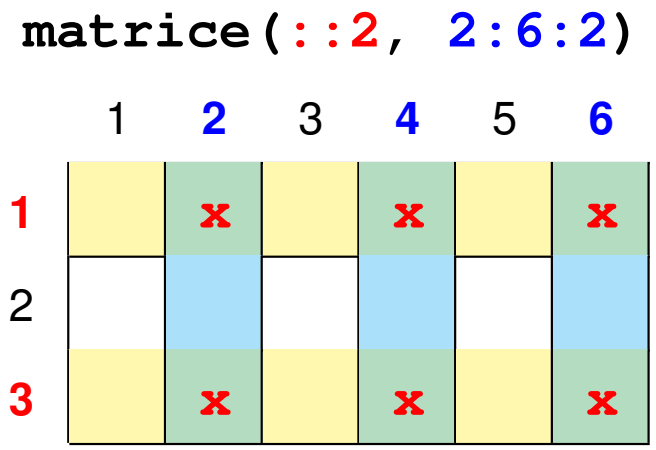
```
REAL, DIMENSION(10,3) :: mat1, mat2, mat3
mat1(:,:) = 2. * mat2(:,:) + 1.     ! scalaires promus tableaux
mat3(:,:) = mat1(:,:) + mat3(:,:)   ! si conformants
mat3(:,:) = COS(mat3(:,:))          ! fonction scalaire "élémentaire"
mat3(:,:) = mat3(:,:) * mat3(:,:)   ! multipl. terme à terme
```

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 (pas=2)
col(:) = mat(2,1:3)           ! col = partie de la ligne 2
```

⚠ début : fin : pas en fortran/numpy mais déb : pas : fin en scilab/octave/matlab





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                 :: nlig, ncol, 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
```
deuxième argument, optionnel : **DIM=**
```
nlig = SIZE(mat1,1) ; ncol = 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
rs = NORM2(vect1)                       ! norme euclidienne (f2008)
```

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**

de 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 Éléments de 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
```

— **WHERE (v > 0)** ! affectation selon un masque

```
  v = log(v)                       ! log protégé
ELSEWHERE
  v = -1.e20
END WHERE
```

— **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
```

FORALL admet un argument optionnel de type masque logique

8.3.5 Ordre des éléments dans les 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
```

$\Rightarrow$ **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 4e 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 1 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**.

⚠ ⇒ En C, extraction de colonne dans une matrice difficile

8.4.2 Sous-tableau 1D avec un pointeur

```c
#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 élém
int i ;

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

```c
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

équivaut à `float tv[100][3];`

8.5 Procédures et tableaux

Langage C	Fortran 90
tableau = pointeur ⇒ cibles modifiables (sauf si **`const`** pour les cibles pointées)	`INTENT(OUT)`, `INTENT(INOUT)` `INTENT(`**`IN`**`)`
passage de tableaux de dimensions fixes (pour le compilateur) en argument	
⇒ 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 ⇒ notation `t[i]` utilisable	**ne pas** transmettre le nb d'éléments ⇒ 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              ! assure la visibilité des interfaces
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) ! boucle sur les lignes
       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 C89

```
#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 fct *
```

```
  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 */
```

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

```c
#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 (juxtaposition de lignes)*/
       printf("%d ", *(tt + i*p + j) ); /* position i*p +j */
    }
    printf("\n") ;
  }
```

```c
}

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 */
```

8.5.4 Tableaux automatiques locaux : C99 et fortran

Langage C99	Fortran 90
Alloués sur la « pile » (stack) ⇒ 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])` ≠ appel : `f(n,t)`	ne pas passer la taille en argument visibilité de l'interface : module + **USE**
récupération de la taille (1re dim.)	
`sizeof(tab)/sizeof(tab[0])`	`SIZE(tab, DIM=1)`

⚠

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**

⇒ **allocation dynamique** sur le « tas » (heap)

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

Passer le nombre d'éléments avant le tableau

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

```c
#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") ;
}
```

```c
// 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  et non (nn, ti[nn])
  print_tab2(nn, 2*nn, ti2); // affichage 2 D
  printf(" tailles en octets d'un scalaire: ti[0] %ld\n", sizeof(ti[0]));
  printf(" tailles en octets de ti (1D) %ld et ti2 (2D) %ld \n",
                        sizeof(ti), sizeof(ti2));
  printf(" dimensions de ti2 (2D) %ld %ld \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
```

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

$\Rightarrow$ 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

$\Rightarrow$ 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

$\Longrightarrow$ **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	
`void *malloc(size_t size)` `void *calloc(size_t nmemb, size_t size)` initialise à **0**	**ALLOCATE**`(Objet(profil),STAT=err)` où `Objet` déclaré **ALLOCATABLE** ou **POINTER**
en cas d'échec de l'allocation	
rend un pointeur **NULL**	`STAT /= 0`
libération de la mémoire allouée	
`void free(void *ptr)` ajouter `ptr=NULL;` pour éviter erreur si autre `free(ptr)`	**DEALLOCATE**`(Objet, STAT=ierr)` puis si pointeur, `Objet => NULL`
annuler aussi les pointeurs qui partagent cette cible	
interrogation sur l'allocation/association	
	ALLOCATED(Objet allouable) $\rightarrow$ booléen **ASSOCIATED**(Objet pointeur) $\rightarrow$ booléen

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 :
  $\Longrightarrow$ 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,
    $\Longrightarrow$ **à convertir explicitement** en pointeur sur le type choisi
  - le pointeur **`NULL`** en cas de problème.
    $\Longrightarrow$ **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 :

De **`ptr[0]`** soit `*ptr`,... jusqu'à **`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 (80 octets)

```
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 :
$\Longrightarrow$ 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**

$\Rightarrow$ ne pas désassocier ce pointeur avant de libérer la zone

sinon zone mémoire **réservée mais inaccessible**

⚠ **fuite de mémoire** (*memory leak*) $\Rightarrow$ 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() ! déassociation 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
```

$\Rightarrow$ 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 déassociation 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)
```

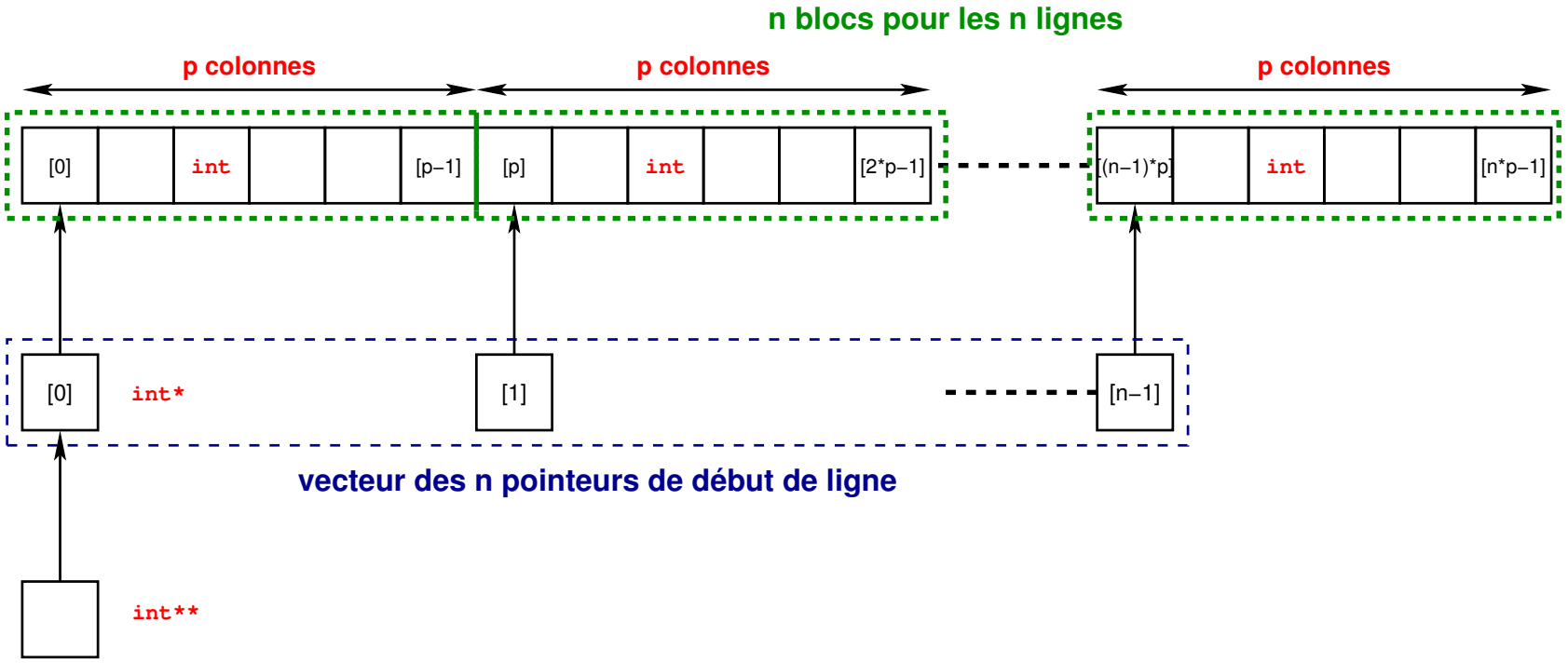
```
    DO i = 1, n
       DO j = 1, m
       ! on pourrait se contenter  de la ligne suivante
       !  c(i, j) = sum(a(i,:) * b(:, j)) ! produit "scalaire"
          c(i, j) = 0
          DO k = 1, p
            c(i, j) = c(i, j) + a(i, k) * b(k, j)
          ENDDO
       ENDDO
    ENDDO
  END SUBROUTINE 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 $\mathtt{n} \times \mathtt{p}$ coefficients comme une **matrice aplatie** **en concaténant les n lignes** (appelée RowMajor dans lapack)



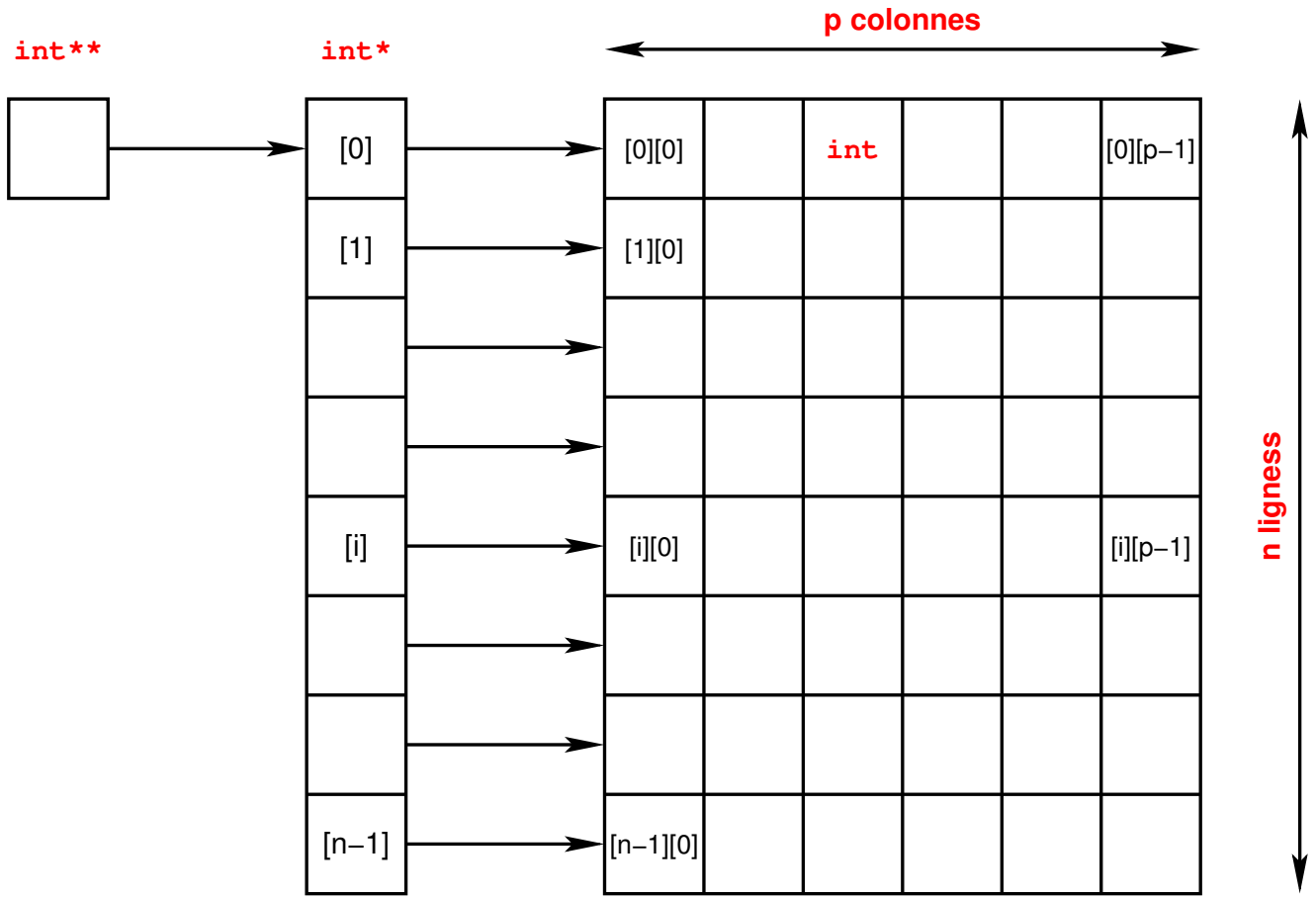
Puis, **structurer** cette zone mémoire linéaire en $\mathtt{n}$ blocs (lignes) de $\mathtt{p}$ cases (colonnes)

$\Rightarrow$ via $\mathtt{n}$ pointeurs de début de ligne $\Rightarrow$ **allouer** aussi un vecteur de pointeurs

$\Rightarrow$ matrice accessible grâce à un **pointeur de pointeurs**

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

Tableau 2D en C = tableau de tableaux ⇒ pointeur de pointeurs de début de lignes



```c
/* 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") ;
    }
}
/*********************************************************/
```

```c
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]) */
```

```c
/* 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 OpenSuse
et sur le serveur appli1

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.$\geqslant$4.6).

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

`tab2 = tab1` $\Longrightarrow$ allocation ou réallocation de `tab2`

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

```
! argument tableau alloué par la procédure (f2003) ! proc+alloc.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
```

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 ')

$\Rightarrow$ 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 ($\geqslant n+1$ avec `\0`)
`size_t strlen(const char *s)`
jusqu'au `'\0'` exclus ($\leqslant n$)
	`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` = **L**exically **G**reater or **E**qual
	résultat booléen

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 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')
```

10.4.2 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 :

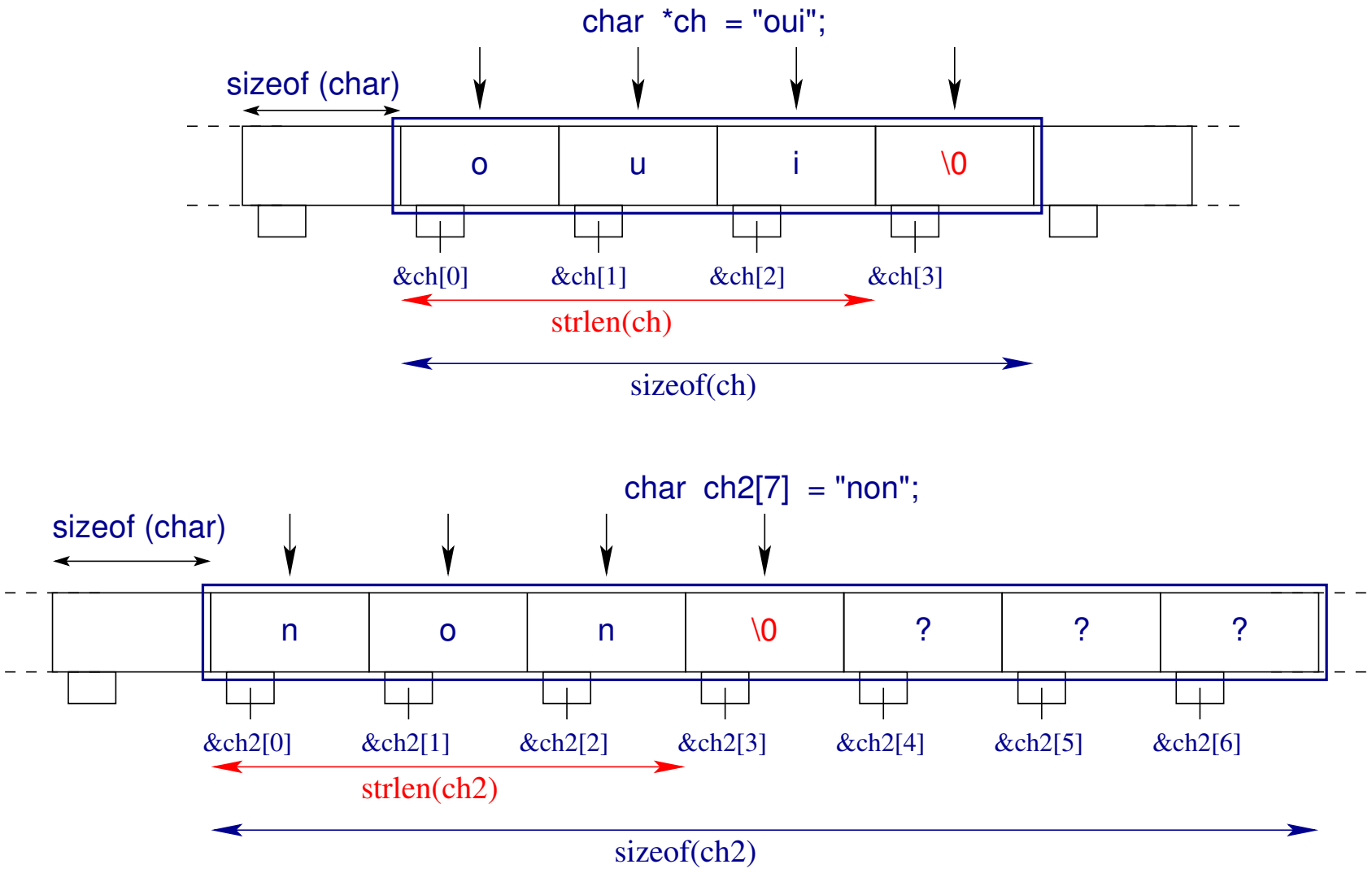
```
sizeof(dest) >= strlen(dest) + strlen(source) + 1
```

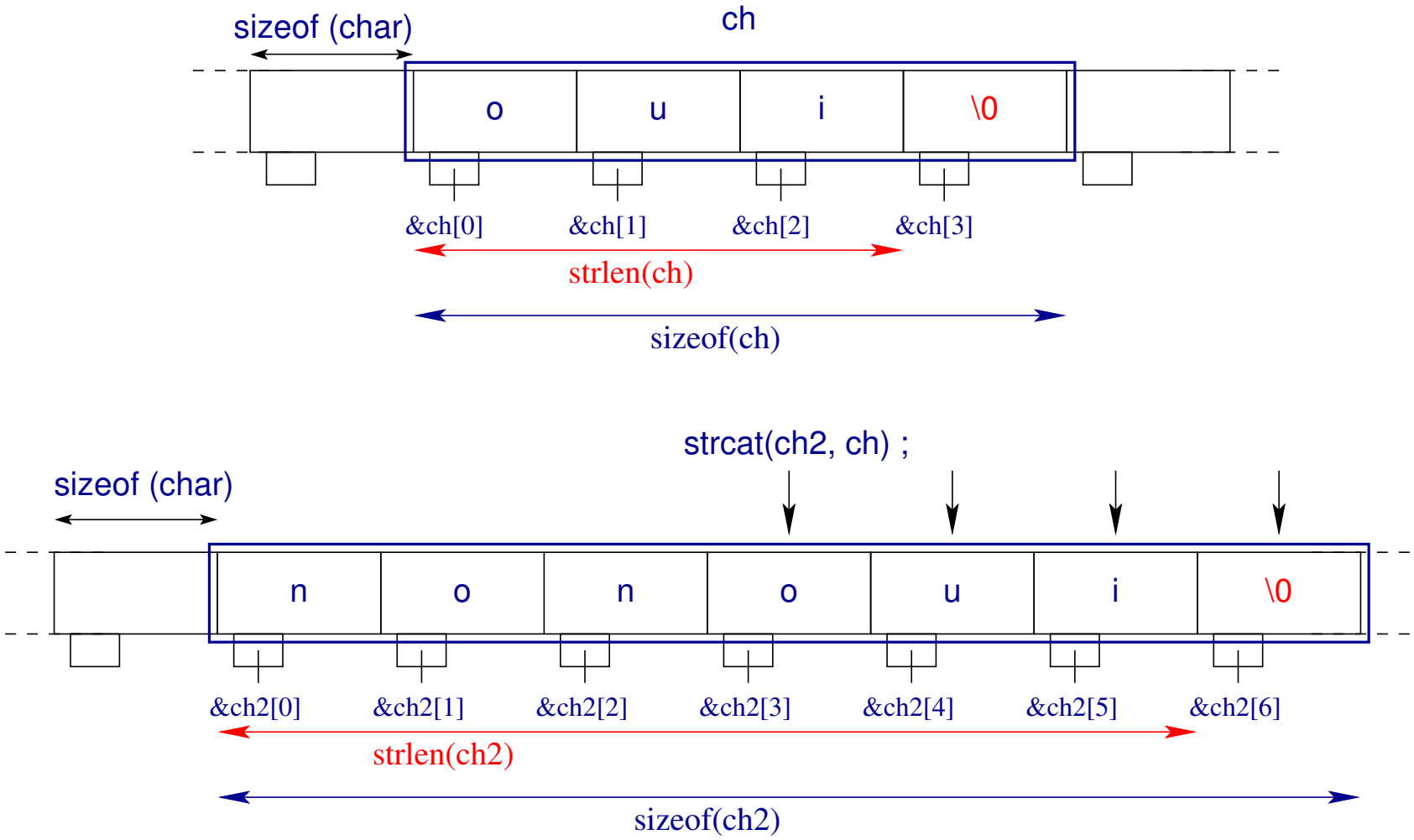
Exemple :

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

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

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


char *ch = "oui";
sizeof (char)
o
u
i
\0
&ch[0]
&ch[1]
&ch[2]
&ch[3]
strlen(ch)
sizeof(ch)
char ch2[7] = "non";
sizeof (char)
n
o
n
\0
?
?
?
&ch2[0]
&ch2[1]
&ch2[2]
&ch2[3]
&ch2[4]
&ch2[5]
&ch2[6]
strlen(ch2)
sizeof(ch2)


sizeof (char)
ch
o
u
i
\0
&ch[0]
&ch[1]
&ch[2]
&ch[3]
strlen(ch)
sizeof(ch)
strcat(ch2, ch) ;
sizeof (char)
n
o
n
o
u
i
\0
&ch2[0]
&ch2[1]
&ch2[2]
&ch2[3]
&ch2[4]
&ch2[5]
&ch2[6]
strlen(ch2)
sizeof(ch2)

10.5 Chaînes de caractères en fortran

10.5.1 Sous-chaînes en fortran

Notation similaire à celle des sous-sections de tableaux

chaine(début:fin) (vide si `fin < début`)

```
character(len=7) :: chaine="bonjour"
character(len=3) :: deb, fin
```

deb = chaine(1:3) $\Longleftrightarrow$ **deb = chaine(:3)**

fin = chaine(5:7) $\Longleftrightarrow$ **fin = chaine(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  ")` $\Rightarrow$ `"  ab cd"`

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

`ADJUSTL("  ab cd  ")` ⇒ `"ab cd    "`

`ADJUSTR("  ab cd  ")` ⇒ `"    ab cd"`

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

REPEAT : **duplication** de chaînes

`REPEAT(" ab ", 3)` ⇒ `" ab  ab  ab "`

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

`INDEX("attente", "te")` ⇒ **3** pour le 1er **te** (**6** si **back=.true.**)

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

`SCAN("aujourd'hui", "ru")` donne **2** pour le 1er **u** (**10** si **back**)

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

`VERIFY("aujourd'hui", "aiou")` donne **3** pour **j** (**9** pour **h** si **back**)

10.5.3 Tableaux de chaînes en fortran

Tableau de chaînes ⇒ rectangulaire ⇒ **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 2e chaîne `tab_ch(2)`

10.5.4 Entrées-sorties de chaînes en fortran

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

Descripteur **An** ⇒ 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

$\Rightarrow$ 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
`FILE *fopen(const char *path, const char *mode)`		`OPEN(UNIT=unit, FILE=filename &` `[, STATUS=mode] [, IOSTAT=i] ...)` `FORM="unformatted"` si binaire `FORM="formatted"` si texte (défaut) `ACCESS="sequential"`(défaut)/`"direct"`
`mode` position `"r"` read début `"w"` write début `"a"` append fin	ajouter `+` si mise à jour, `b` si binaire	`mode` = `"old"` `"new"` `"replace"`
rend `NULL` si erreur		`IOSTAT`$\neq$ 0 si erreur

fermeture (et vidage du tampon)	
`int fclose( FILE *stream)`	`CLOSE(unit[,IOSTAT=ivar,...])`

11.2 Entrées-sorties formatées (fichiers codant du texte)

3 fonctions en C, 1 fonction en fortran

écriture	C	fortran 90
stdout	`printf(...)` = `fprintf(stdout, )`	`write(*, ...)` ou `print ...,`
fichier	`fprintf(FILE* stream, ...)`	`write(unit, ...)`
chaîne	`sprintf(char* string, ...)`	`write(chaine, ...)`

lecture	C	fortran 90
stdin	`scanf(...)` = `fscanf(stdin, )`	`read(*, ...)` ou `read ...,`
fichier	`fscanf(FILE* stream, ...)`	`read(unit, ...)`
chaîne	`sscanf(char* string, ...)`	`read(chaine, ...)`

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	

`int fscanf(FILE *stream,`
`  const char *format,`
`  liste de pointeurs)`
 	
valeur de retour = nb de conversions
	`READ(unit, format  &`
	`  [, IOSTAT=ok...]) &`
	`  liste de variables`
	
	IOSTAT = statut d'erreur (0 si correct)

sortie	

`int fprintf(FILE *stream,`
`  const char *format,`
`  liste d'expressions)`
 	
valeur de retour = nb de caractères
	`WRITE(unit, format  &`
	`  [, IOSTAT=ok...]) &`
	`  liste d'expressions`
	
	IOSTAT = statut d'erreur (0 si correct)

Séparation en lignes

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

11.3 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	%w**o**	**O**w
hexadécimal	%w**x**	**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++ : `cin`/`cout`)	**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 ≠ 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

11.4 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,dett
      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 pa
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
```

11.5 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
	$\Rightarrow$ format réexploré $\Rightarrow$ 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

$\Rightarrow$ 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 $\rightarrow$ **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** :

$\rightarrow$ objets dynamiques complexes tels que arbres, **listes chaînées**

$\Rightarrow$ 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 $\Rightarrow$ 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	

`struct point {`
`  int no ; // 1er champ`
`  float x ; // 2e champ`
`  float y ; // 3e champ`
`  };`
	`TYPE point`
	`  integer :: no ! 1re composante`
	`  real :: x ! 2e composante`
	`  real :: y ! 3e composante`
	`END TYPE point`

`typedef struct point spoint;` `spoint` synonyme de `struct point`	
Où les déclarer ?	
dans un fichier d'entête puis `#include`	dans un module (avant `CONTAINS`) puis `USE`

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`	
`struct point debut, fin ;` ou `spoint debut, fin;`	`TYPE (point) :: debut, fin`
Initialisation via un constructeur	
`struct point fin={9,5.,2.};`	`TYPE(point)::fin=point(9,5.,2.)`
Implémentation des structures	
Contraintes d'alignement en mémoire $\Rightarrow$ compléter parfois avec des octets de remplissage	
La taille de la structure n'est pas toujours la somme des tailles	
utiliser `sizeof` si allocation	
ordre des champs respecté	**ordre des compos. modifiable sauf si attribut `SEQUENCE`**

12.3 Manipulation des structures

Langage C	Fortran 90
Accès aux champs d'une structure / composantes d'un type dérivé	

`debut.no = 1 ;`
`debut.x = 0. ;`
`debut.y = 0. ;`
	`debut%no = 1`
	`debut%x = 0.`
	`debut%y = 0.`

Affectation globale (même type)	
`fin = debut ;` constructeur interdit (sauf initialis.)	`fin = debut` constructeur `fin=point(9,5.,2.)`
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
```

```c
#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	

`struct point3d {`
`  char nom[5];`
`  float coord[3];`
`} ;`
`struct point3d p1;`
	`TYPE point3d`
	`  CHARACTER(len=4)  :: nom`
	`  REAL, DIMENSION(3):: coord`
	`END TYPE point3d`
	`TYPE(point) :: p1`

ordonnée de p1	
`p1.coord[1]`	`p1%coord(2)`
Tableaux de structures	
`struct point courbe[9] ;`	`TYPE(point),dimension(9)::courbe`
abscisse du premier élément du tableau	
`courbe[0].x = 2. ;`	`courbe(1)%x = 2.`

12.5 Structures/types dérivés, pointeurs et procédures

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

```
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
```

```c
#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;
}
```

```c
void sym(spoint m, spoint *n); /* passer le pointeur pour pouvoir */
void sym(spoint m, spoint *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){
  spoint a = {5, 1. ,-2.}; /* définition 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); /* 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**
`struct point{`	`type point`
`  float x;`	`  real :: x`
`  float y;`	`  real :: y`
`  struct point *next;`	`  type(point), pointer :: next`
`};`	`end type point`

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

$\Rightarrow$ 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)

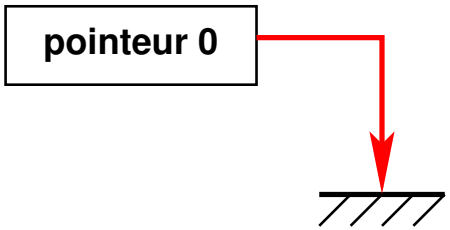
En fin de boucle : le dernier élément doit pointer vers **`NULL`**.

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.)

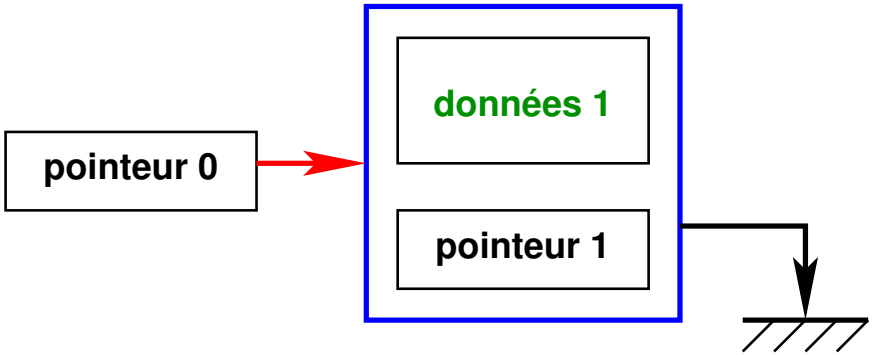
mais ne pas perdre le chaînage !



Liste vide :

pointeur de tête nul

Construction d'une liste simplement chaînée : deux premiers éléments

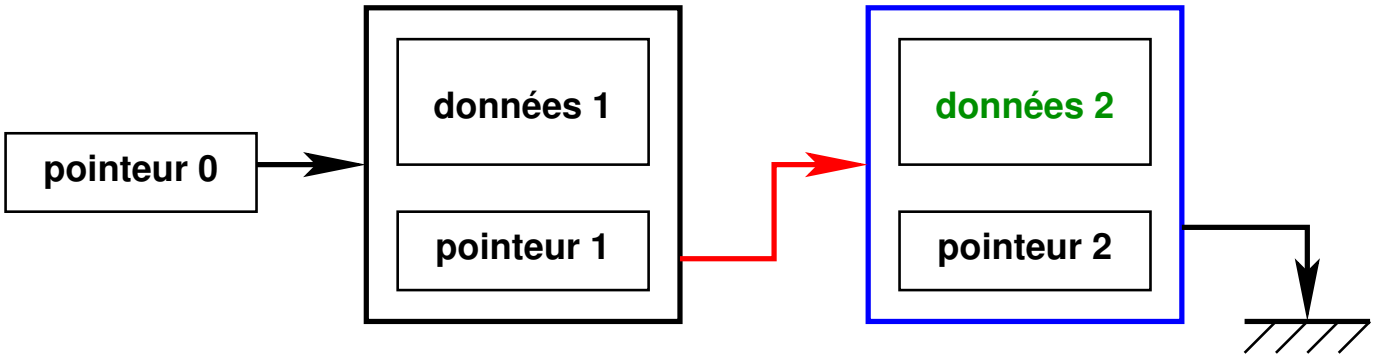


Allocation élément 1

Tête pointe vers él. 1

Affectation des données 1

Pointeur 1 nul



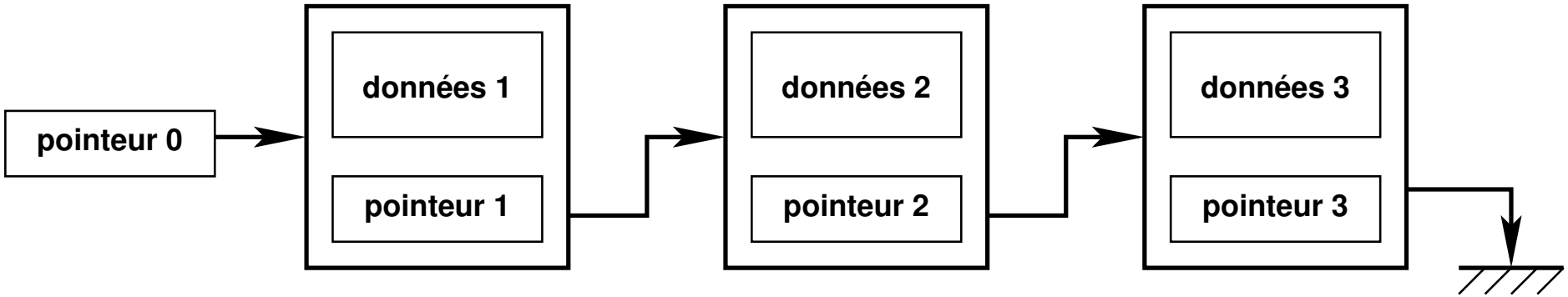
Allocation élément 2

Pointeur 1 vers él. 2

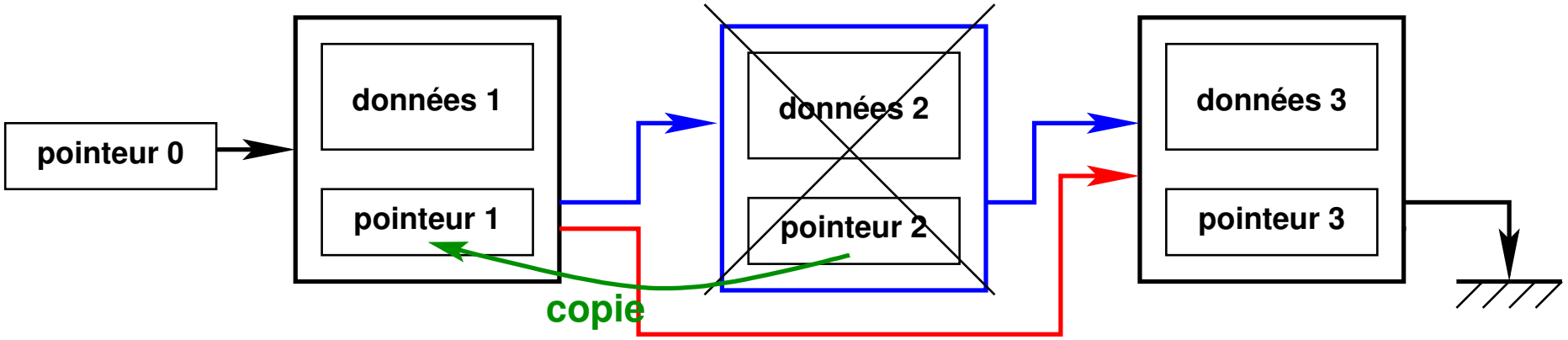
Affectation des données 2

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*)

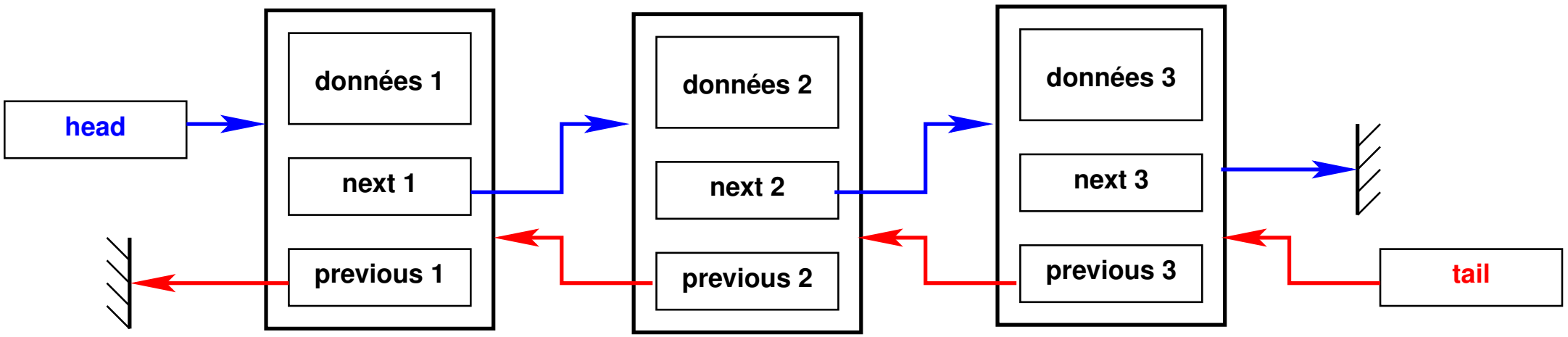
Problème de la liste simplement chaînée : les noeuds de fin de liste ne sont accessibles qu'en parcourant tout le début.

$\Rightarrow$ 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	
pointeur vers le tableau	tableau allouable

`struct point_ndim {`
`  int n;` numéro
`  int dim;` nb de dim.
`  float* coord;` vecteur
`} ;`
`struct point_ndim a, b;`
	`TYPE point_ndim`
	`  INTEGER :: n` numéro
	`  REAL, DIMENSION(:), &`
	`  ALLOCATABLE :: coord`
	`END TYPE point_ndim`
	`TYPE(point_ndim) :: a, b`

Allocation	

à prendre en charge par l'utilisateur
`a.n = 5;`
`a.dim = 2;`
`a.coord=calloc(2, sizeof float);`
`a.coord[0] = 1.;`
`a.coord[1] = -3.;`
	`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.])`

Libération	
à prendre en charge sinon fuite de mémoire	automatique en sortie de portée

Langage C	Fortran 2003
Affectation globale	

copie superficielle
`struct point_ndim a, b;`
`b = a;`
⚠ ne recopie que les champs (donc le pointeur)
mais pas les variables pointées
⇒ à prendre en charge
par une fonction de recopie avec allocation
	copie profonde
	`TYPE(point_ndim) :: a, b`
	`b = a`
	recopie les composantes donc les valeurs
	en réallouant à la volée si nécessaire
	`b = point_ndim(7, [-1.,3., 2.])`
	`b = a` réalloue à la nouvelle taille

NB : pas de tableaux automatiques dans les structures en C	

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)
$\Longrightarrow$ 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 et les tableaux

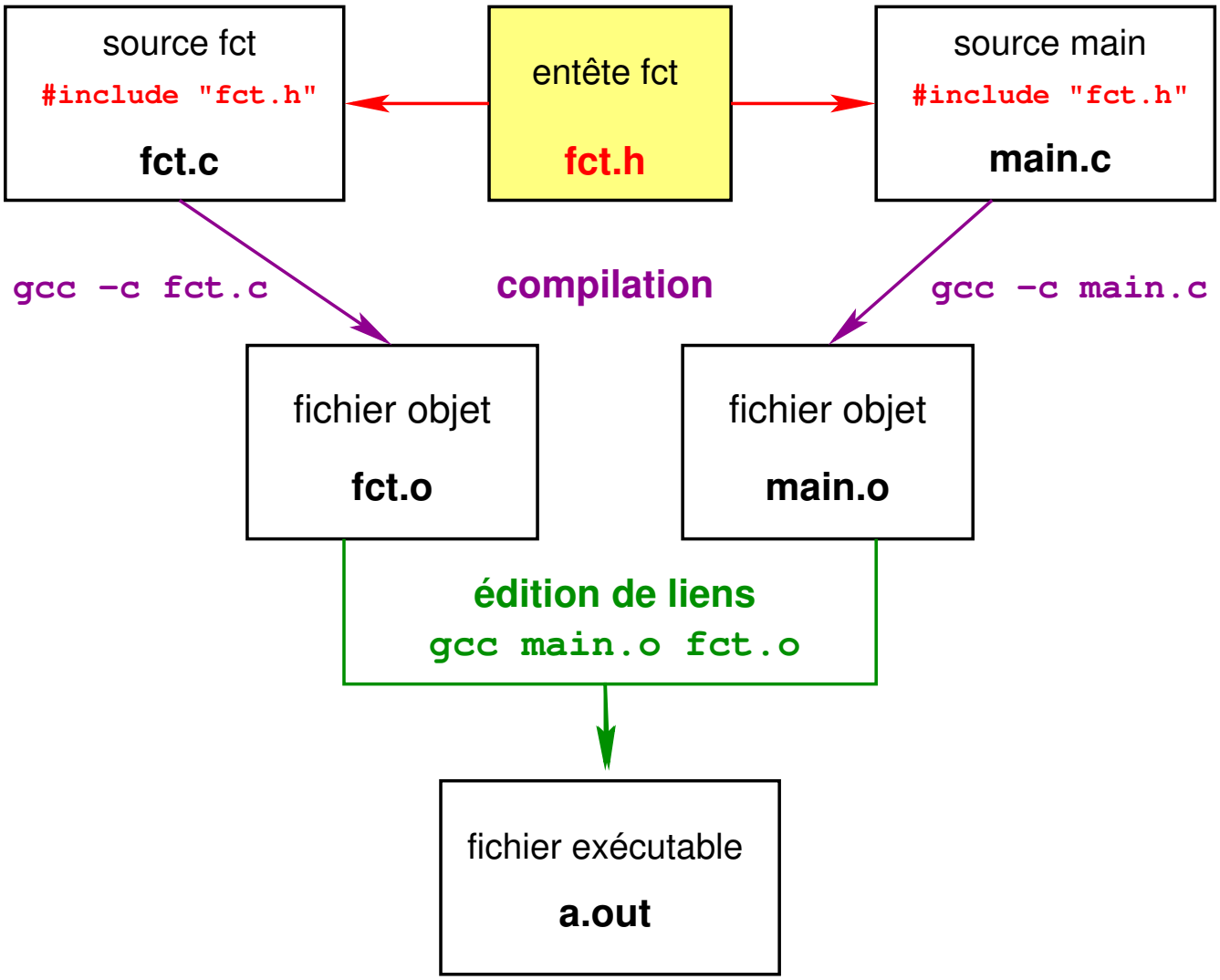
Première solution : dupliquer l'information sur l'interface en la déclarant au niveau des procédures qui l'utilisent.

Fortran	langage C
déclaration d'interface	déclaration de prototype

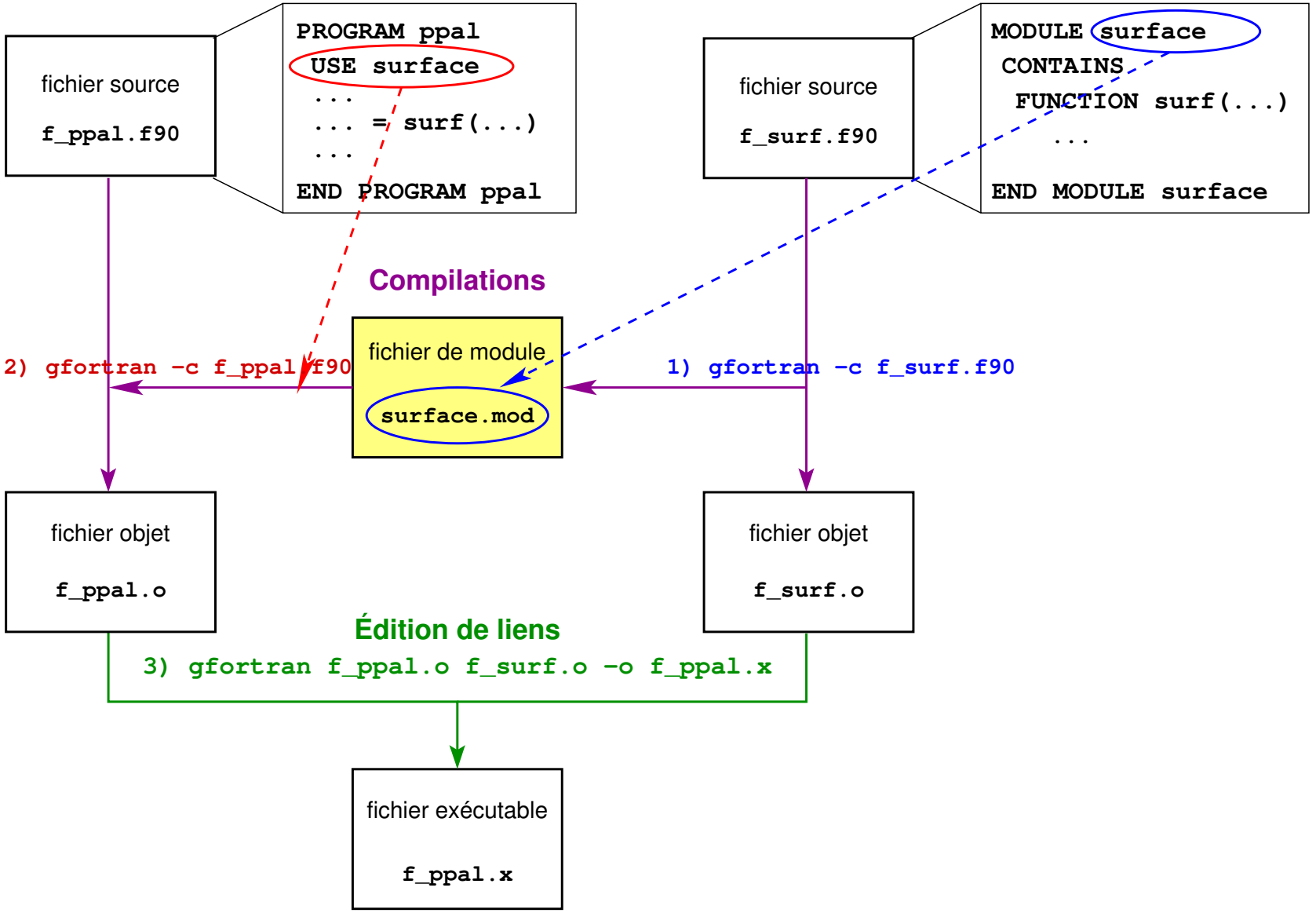
Mais risque d'incohérence entre **déclaration** et **définition** en particulier dans la phase de développement où la liste des arguments peut évoluer.

$\Rightarrow$ **Solution plus robuste**

Fortran	langage C
Encapsuler les procédures dans des **modules** : le compilateur crée alors des fichiers `.mod` décrivant l'interface. L'instruction **`USE`** permet de relire cette interface quand on compile les appelants.	Déclarer les prototypes dans les fichiers d'**entête `*.h`** et les inclure à la fois : — dans la définition — et dans les fonctions appelantes.
$\Rightarrow$ Compiler les modules utilisés avant de compiler les appelants	$\Rightarrow$ Éviter les déclarations multiples si plusieurs inclusions **`#ifndef`** ... **`#define`** ... **insérer ici les prototypes** **`#endif`**



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 entre déclaration et définition,
- **les fichiers où la fonction est appelée**, pour assurer la cohérence entre déclaration et 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

```c
/* 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);
}
```

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

```c
/* 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
```

$\Rightarrow$ 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
```

$\Rightarrow$ 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 ou interfaces 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 ⇒ extension `.a`)
édition de liens **statique** : objets intégrés à l'exécutable
⇒ exécutable autonome mais plus volumineux

— bibliothèque **dynamique partageable** (*shared object* ⇒ extension `.so`)
édition de liens **dynamique** : objets chargés plus tard en mémoire lors de l'exécution ⇒ 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
```

13.5.3 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`.

13.5.4 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
$\Rightarrow$ `gcc+mni` est un alias vers :

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

où `/home/lefrere/M1/OpenSuse/include/` contient **`mnitab.h`**
et `/home/lefrere/M1/OpenSuse/lib/` contient **`libmnitab.a`**

⚠ Les objets ne sont pas portables $\Rightarrow$ autres versions pour **`sappli1`** 64 bits dans `/home/lefrere/M1/sappli1-64/lib/` et idem pour `include/`
et alias **`gcc64+mni`** avec options **`-L`** et **`-I`** associées

13.6 Génération d'un fichier exécutable avec `make`

13.6.1 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

13.6.2 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)
```

$\Rightarrow$ nécessite d'intégrer des commandes shell dans le **`makefile`**

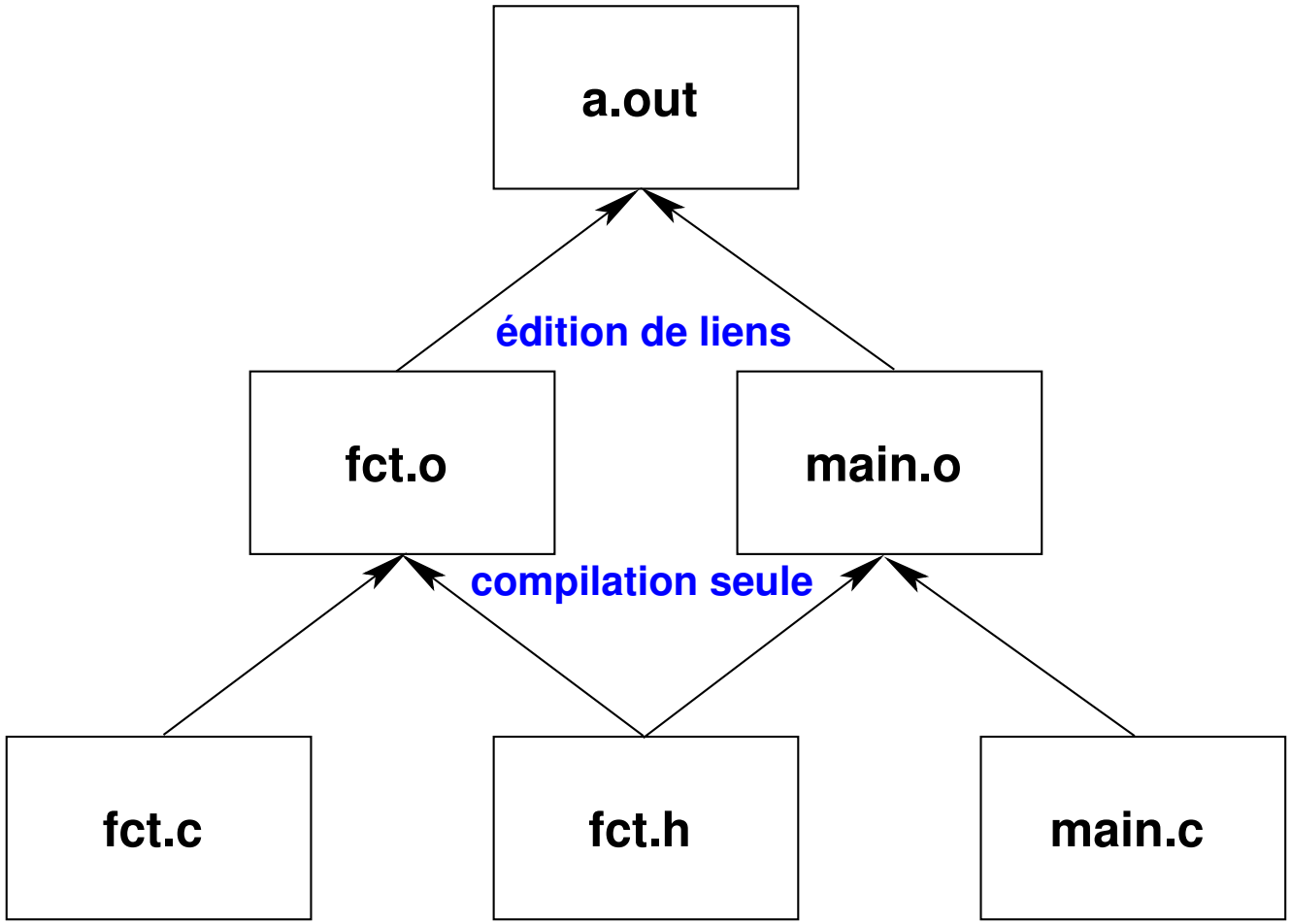
Le fichier **`makefile`** est construit à partir de l'arbre des dépendances.

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

`gfortran -cpp -M fichier.f90` ($\geqslant$ v4.6) (nécessite les `.mod`)

13.6.3 Exemple élémentaire de `makefile` en C

Arbre des dépendances (exploré **récursivement** 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

# cible ménage = suppression des fichiers reconstructibles
clean :
__TAB__/bin/rm -f a.out *.o
```

En guise de conclusion

Fortran et C : deux langages de haut niveau finalement assez proches malgré les différences de syntaxe.

Fortran plus dédié au calcul numérique, mais C permet de mieux comprendre les mécanismes sous-jacents en accédant à des aspects plus bas niveau (adresses).

Différences fortes apparues avec fortran 90 au niveau des tableaux et de leur traitement global avec opérateurs et fonctions.
Pas de manipulation globale des tableaux en C, mais tableaux automatiques à déclaration tardive 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.

Table des matières