UPMC

Master P&A/SDUEE

UE MNI (4P009)

Méthodes Numériques et Informatiques

Fortran 95/2003 et C

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

2015–2016

Table des matières

1 Introduction

1.1 Langage compilé et langage interprété

Langage **compilé**	Langage **interprété**
C, C++, **fortran**	**shell**, **python**, perl, php, scilab
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 n'est pas produit, donc pas d'exécutable !	les instructions avant l'erreur sont exécutées
Compilé + interprété, par exemple **java** compilation $\rightarrow$ bytecode portable interprété par JVM (machine virtuelle java)	

1.2 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

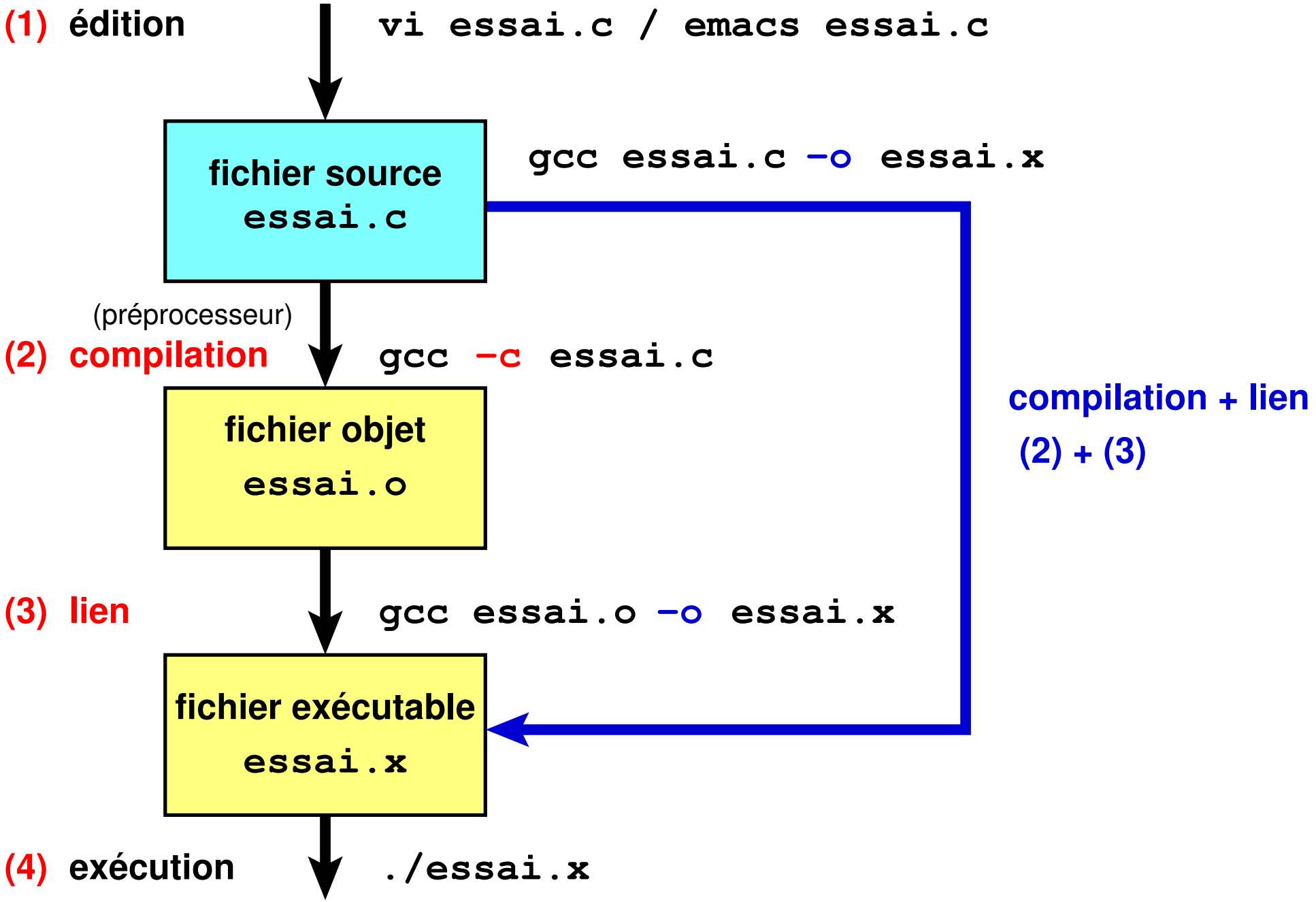
— Fichier **source** (texte) de suffixe `.c` en C (`.f90` en fortran 90)
écrit au moyen d'un **éditeur de texte**

— 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 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, changement de type, ...) : **`-Wall`**

→ vérification des passages de paramètres

(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 : **fortran 2003** standard adopté

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

— futur standard fortran 2015 en cours de révision

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

— base d'autres langages dont le **C++** (premier standard en 1998)
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, ...	
généricité des fonctions mathématiques	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**	plus généraliste ex. : écriture de systèmes d'exploitation
tableaux multidimensionnels et fonctions associées (cf. matlab et scilab)	

Exemple : appel à une procédure de la bibliothèque LAPACK

pour le calcul des éléments propres d'une matrice $m \times m$ réelle.

Langage C

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

12 arguments + status de retour optionnel

Langage Fortran 90

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

5 arguments seulement + status optionnel passé par mot-clef

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

Nombre de bits fixe (typage statique) $\Rightarrow$ **domaine couvert limité**

Mais distinguer :

— les **entiers représentés exactement**

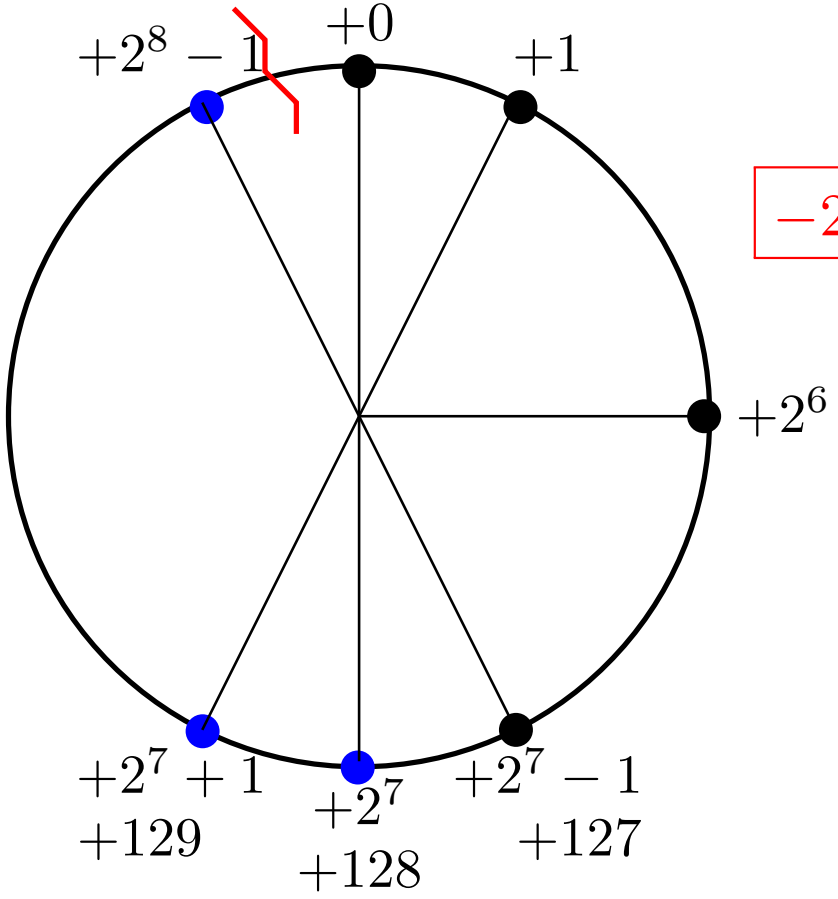
— les **réels représentés approximativement** en virgule flottante

⚠ $\Rightarrow$ ne jamais compter avec des réels

2.1.1 Domaine des entiers signés

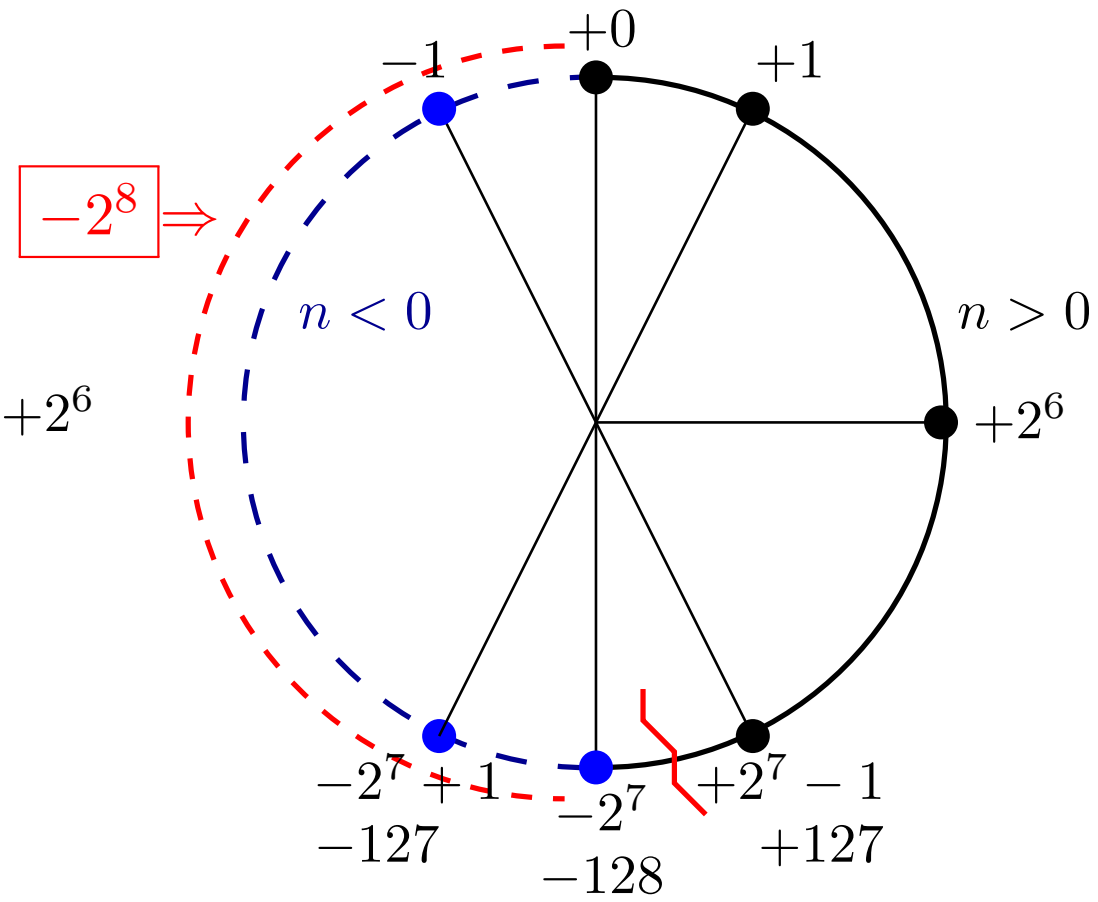
Exemple introductif des entiers sur **1 octet** (8 bits) : $2^8 = 256$ valeurs possibles

taille	non signés	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 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
		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`**

⚠ Dépassement de capacité en entier positif ⇒ passage en négatif

→ en fortran, choix des variantes (**`KIND`**) d'entiers selon le domaine (*range*) par la fonction **`SELECTED_INT_KIND(...)`**

→ en C89, les tailles des entiers dépendent du processeur

→ C99 : types entiers étendus à nb d'octets imposé, par exemple : **`int32_t`**

2.1.3 Domaine et précision des flottants

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

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

exposant

$0.\underbrace{1234}_{\textbf{mantisse}} \times 10^{\overbrace{-1}^{\textbf{exposant}}}$

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 $1 + \varepsilon$ soit le successeur de 1

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

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

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

— Octaves en progression géométrique de raison 2

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

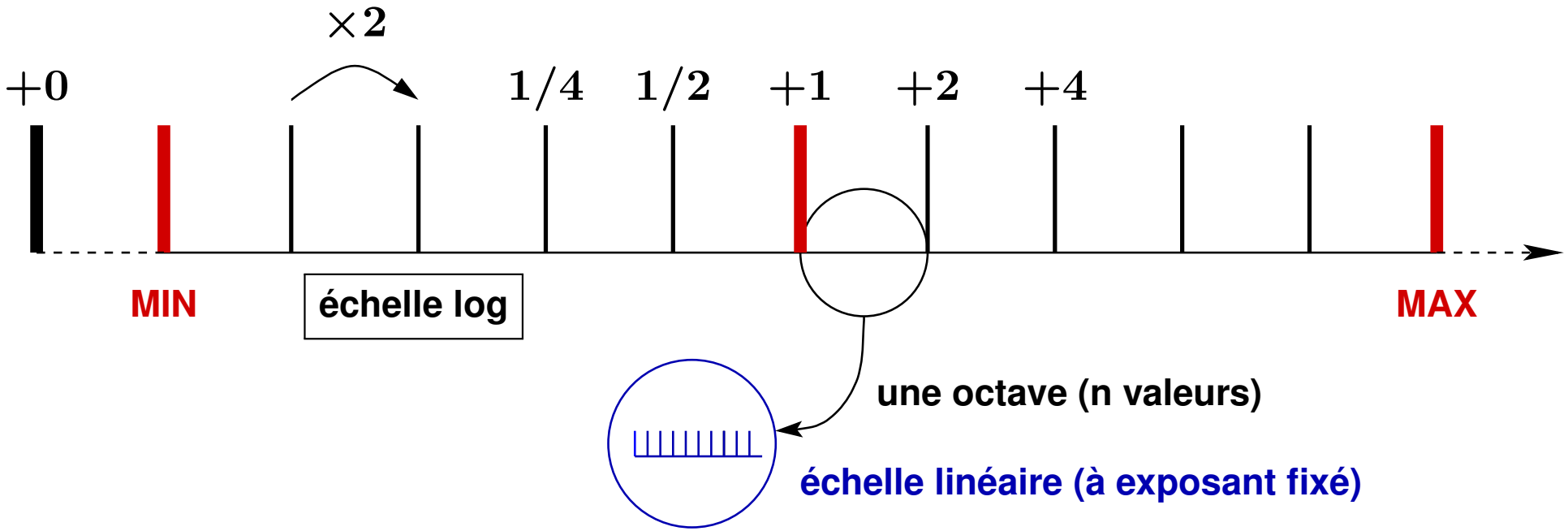
Flottants sur 32 bits : $q =$ **8 bits d'exposant** donc 254 octaves

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.

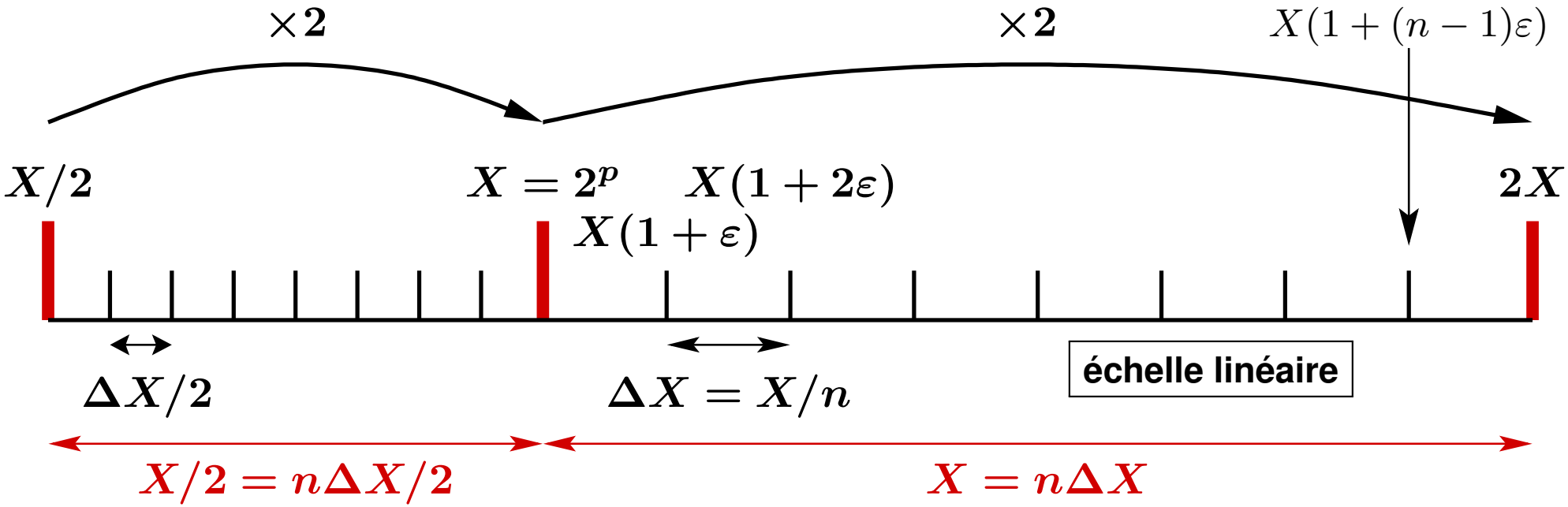


Domaine des flottants positifs normalisés (**échelle log**)

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

	fortran	C	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(1.d0)`	`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 du domaine de x (plus petite valeur absolue)
TINY (x)	plus petite valeur positive représentable dans le type de x
HUGE (x)	plus grande valeur positive représentable dans le type de x

Portabilité numérique du code

$\Rightarrow$ 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)	`real(kind=10/16)`
////C99 **`complex`** **`#include <tgmath>`**	complexe + variantes	**`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"	chaînes	'chaine' "s'il"
17	entier court	17_**2**
17	**entier décimal**	17
⚠ **0**21 (attention)	entier 17_{10} en octal	**O'**21'
0x11	entier en hexadécimal	**Z'**11'
17**L**	entier long	17_**8**
-47.1**f** -6.2e-2**f**	réel simple précision	**-47.1 -6.2e-2**
-47.1 -6.2e-2	double précision	47.1_**8** -6.2e-2_**8**
-47.1**L** -6.2e-2**L**	double précision long	47.1_**16** -6.2e-2_**16**
///// sauf **I** C99 : 2.3-I*.5	complexe	**(**2.3, -5.**)**

2.4 Déclarations des variables

Typage statique => déclarer le type de chaque variable

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 !

⇒ 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)`

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

```
float: 4 octets              int: 4 octets
n (int)    =  123456789      exact
nf (float) = 123456792       approché
b (float) =0.123456791
```

3.2 Opérateurs algébriques

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

Opérations binaires ⇒ même type pour les opérandes (sauf réel`**`entier fortran)

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.

`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**.

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

3.10 Opérateurs & et * en C

&objet $\Rightarrow$ adresse de l'objet

***pointeur** $\Rightarrow$ 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 `%d`, `%g` ou `%s` ... selon le type pour chaque variable (gabarit optionnel)	format **libre** (`*`) le plus simple mais on peut préciser
spécifier `\n` en sortie pour changer de ligne	forcer par `/` dans le format **changement d'enregistrement** à chaque ordre `READ` ou `WRITE`

4.1 Introduction aux formats d'entrée–sortie

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

	c	fortran 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`

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`**...)

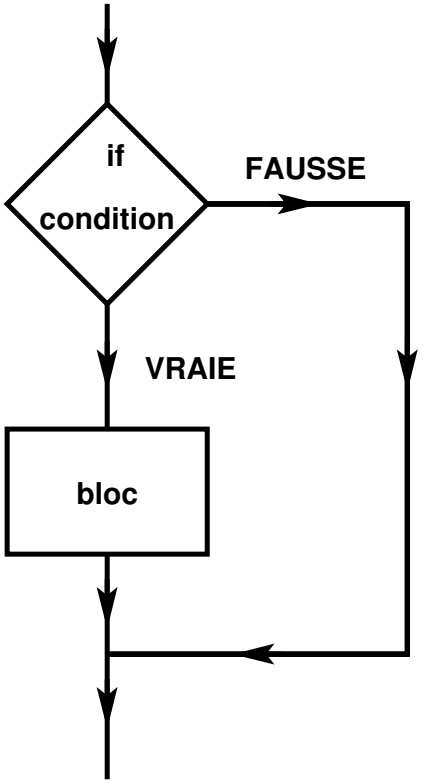
— branchements (**`cycle`** ou **`continue`**, **`exit`** ou **`break`**, ...)

$\Rightarrow$ 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`

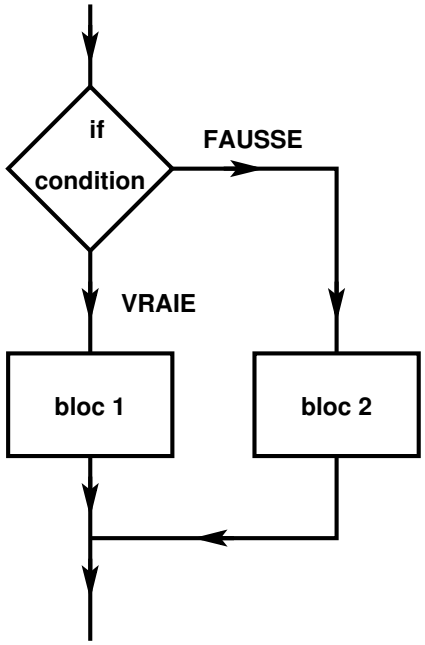


c	fortran 90
if (expression**)** instruction	**if (**expr. log.**)** instruction

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

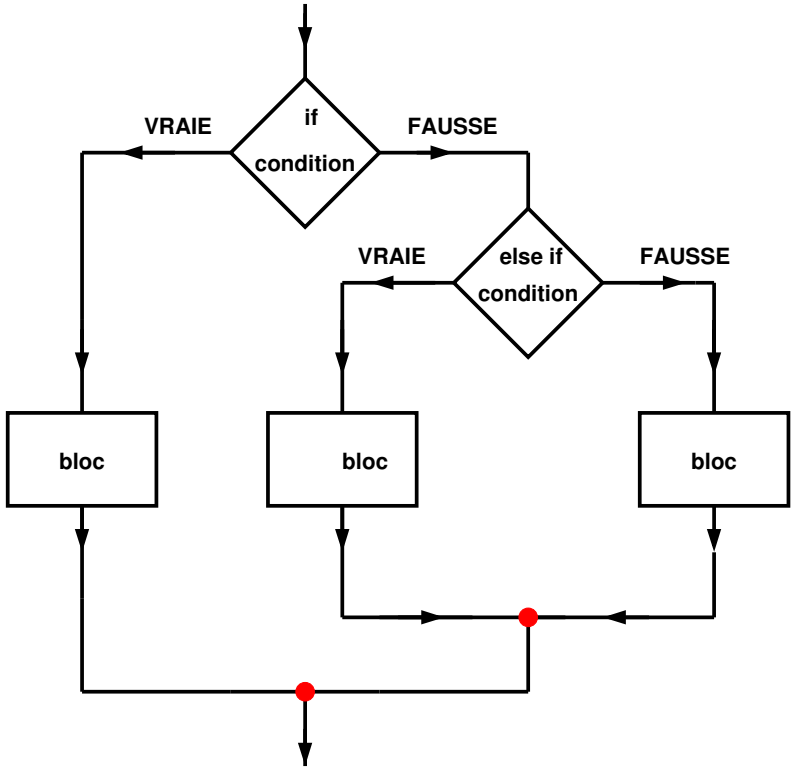
5.1.3 Exemples d'alternative if ... else

```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) ;
}
```

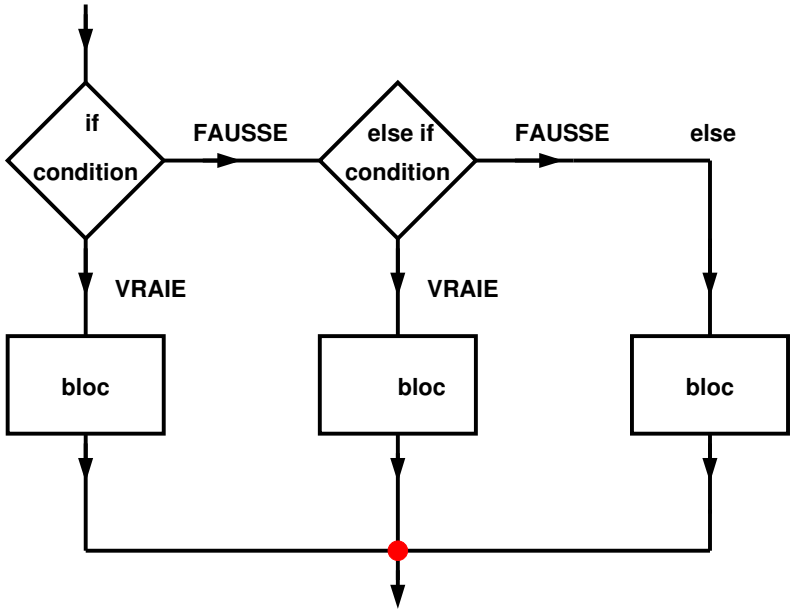
```
! structure   if then ... else ... endif
! affichage du max de deux nombres
PROGRAM alternative
IMPLICIT NONE
INTEGER :: i, j, maxij
WRITE(*,*) "entrer i et j (entiers)"
READ(*,*) i, j
IF (i >= j) THEN
  WRITE(*,*) "i >= j "
  maxij = i                    ! bloc d'instructions
ELSE
  maxij = j  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

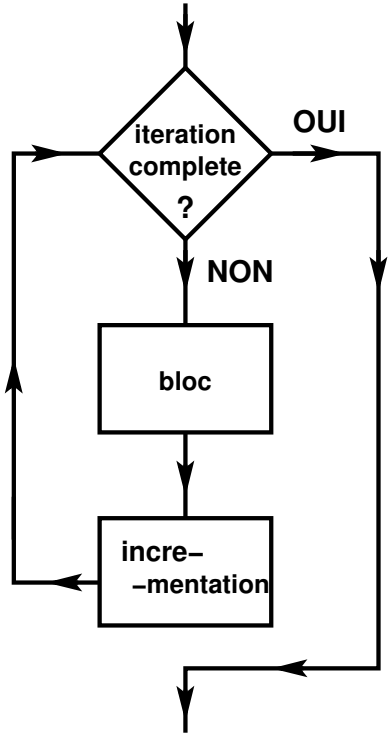
```c
/* structure case sans break
 * pour "factoriser des cas"
   et les traiter en commun */
switch (c)  /* c de type char */
  {
  case '?' :
  case '!' :
  case ';' :
  case ':' :
    printf(" ponctuation double \n") ;
    break ;        /* à la fin des 4 cas */
  default :
    printf(" autre caractère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

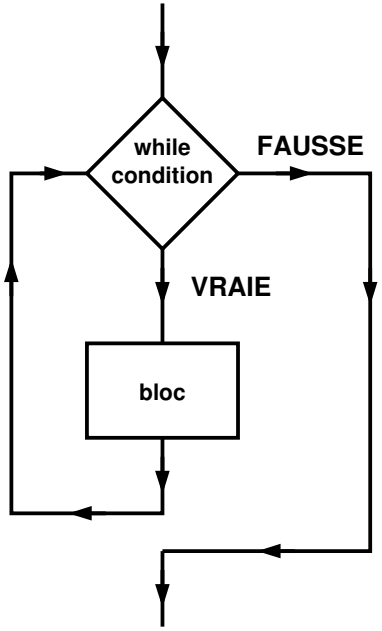
5.3 Structures itératives ou boucles

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 **} 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 ! */
exit(EXIT_SUCCESS) ;
```

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
```

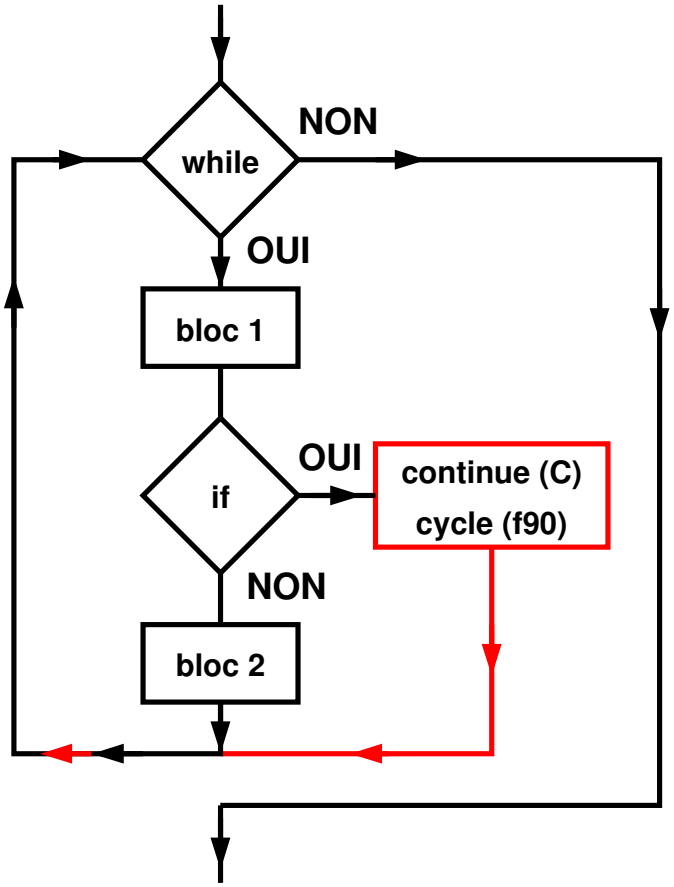
5.4 Branchements ou sauts

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.

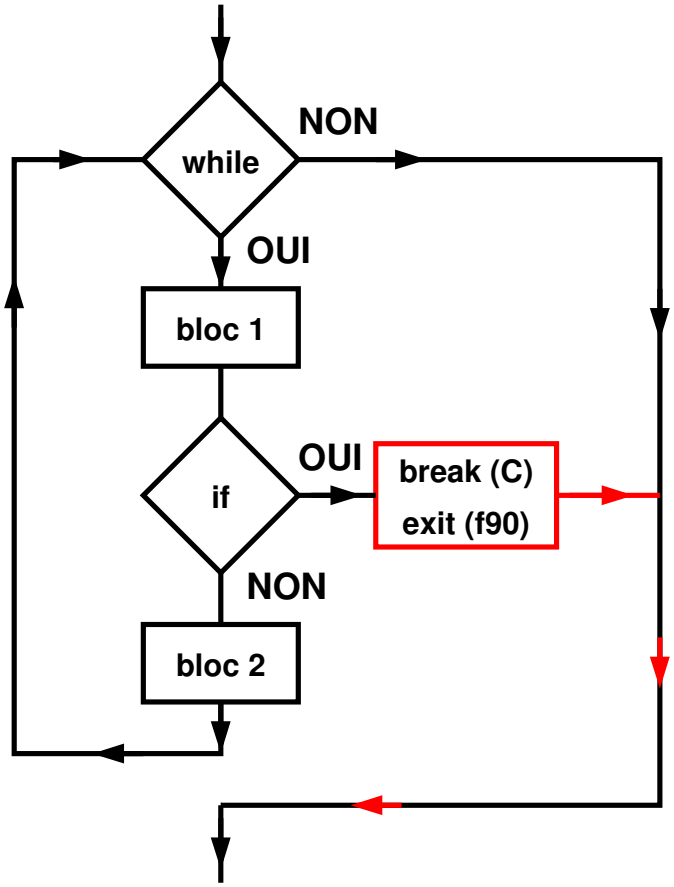
Rebouclage anticipé

`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) ;
 }
```

```fortran
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) ;
   }
```

```
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 :

$\Rightarrow$ faire un intervenir un niveau supplémentaire de paramétrage dans la manipulation des données : action indirecte sur une variable

$\Rightarrow$ créer ou supprimer des variables lors de l'exécution : allocation dynamique

— **différences** notables entre pointeurs en **C** et en **fortran**

indispensables en C	absents en fortran 77,
pour les arguments des fonctions	mais introduits en fortran 90/95
et les tableaux dynamiques	et étendus en fortran 2003
stockent des **adresses** des variables	pas d'accès aux adresses

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

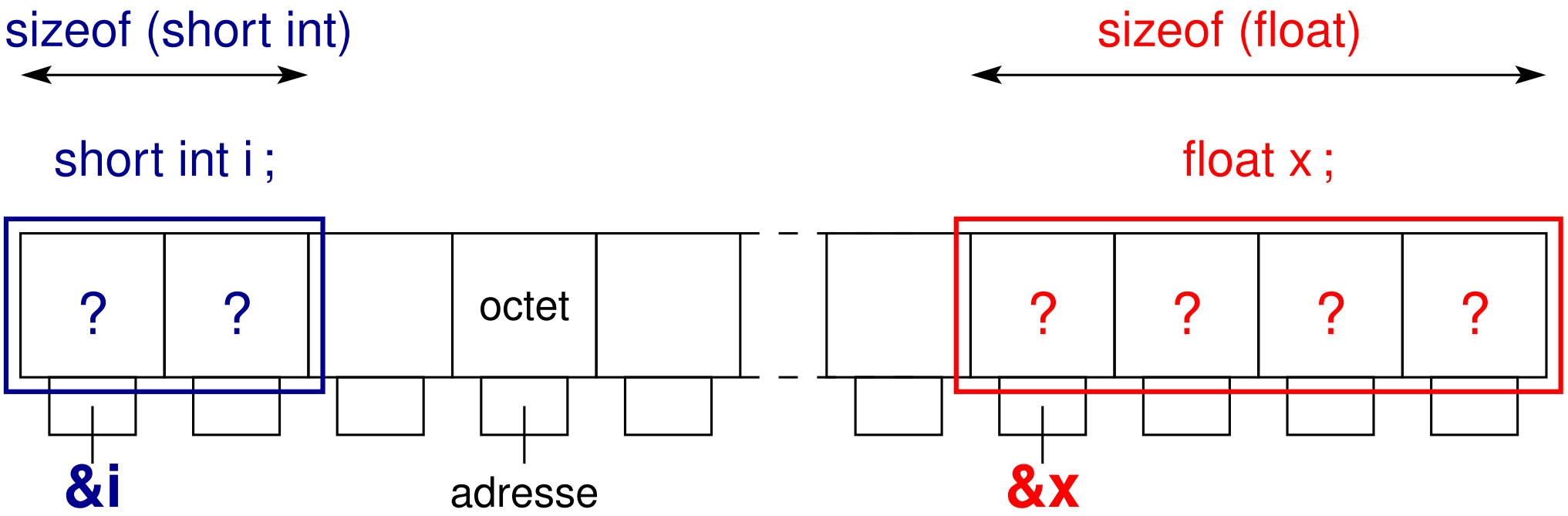
6.2 Pointeurs et variables : exemple du C

Déclarer une variable d'un type donné =

réserver une zone mémoire dont

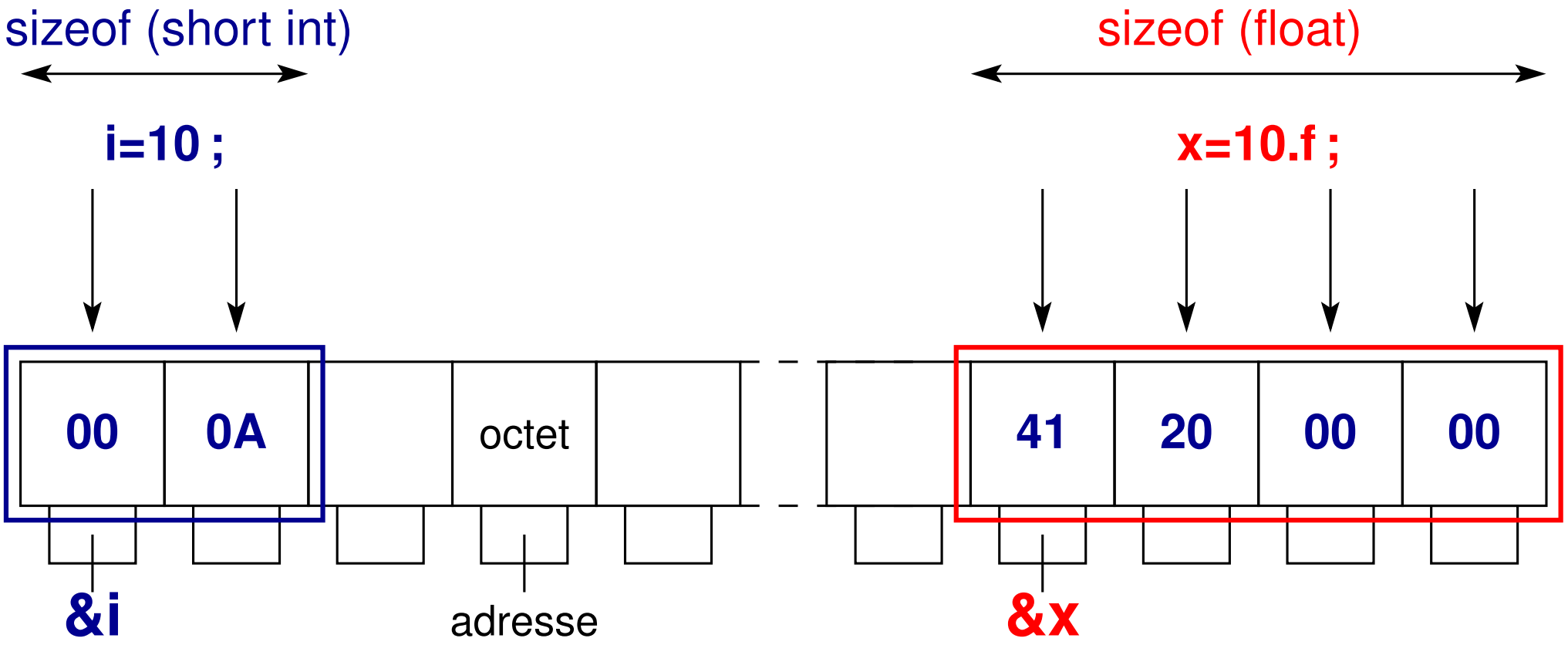
la taille (`sizeof` en C) dépend du type

et le codage est fixé par le type



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

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

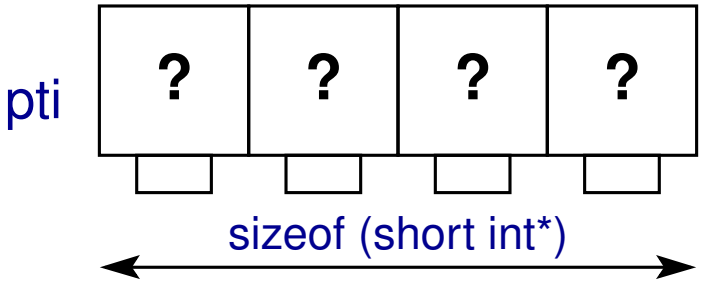


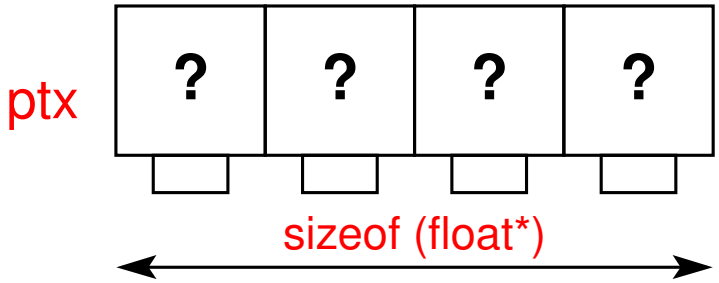
Déclarer une variable pointeur vers un type donné =

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**

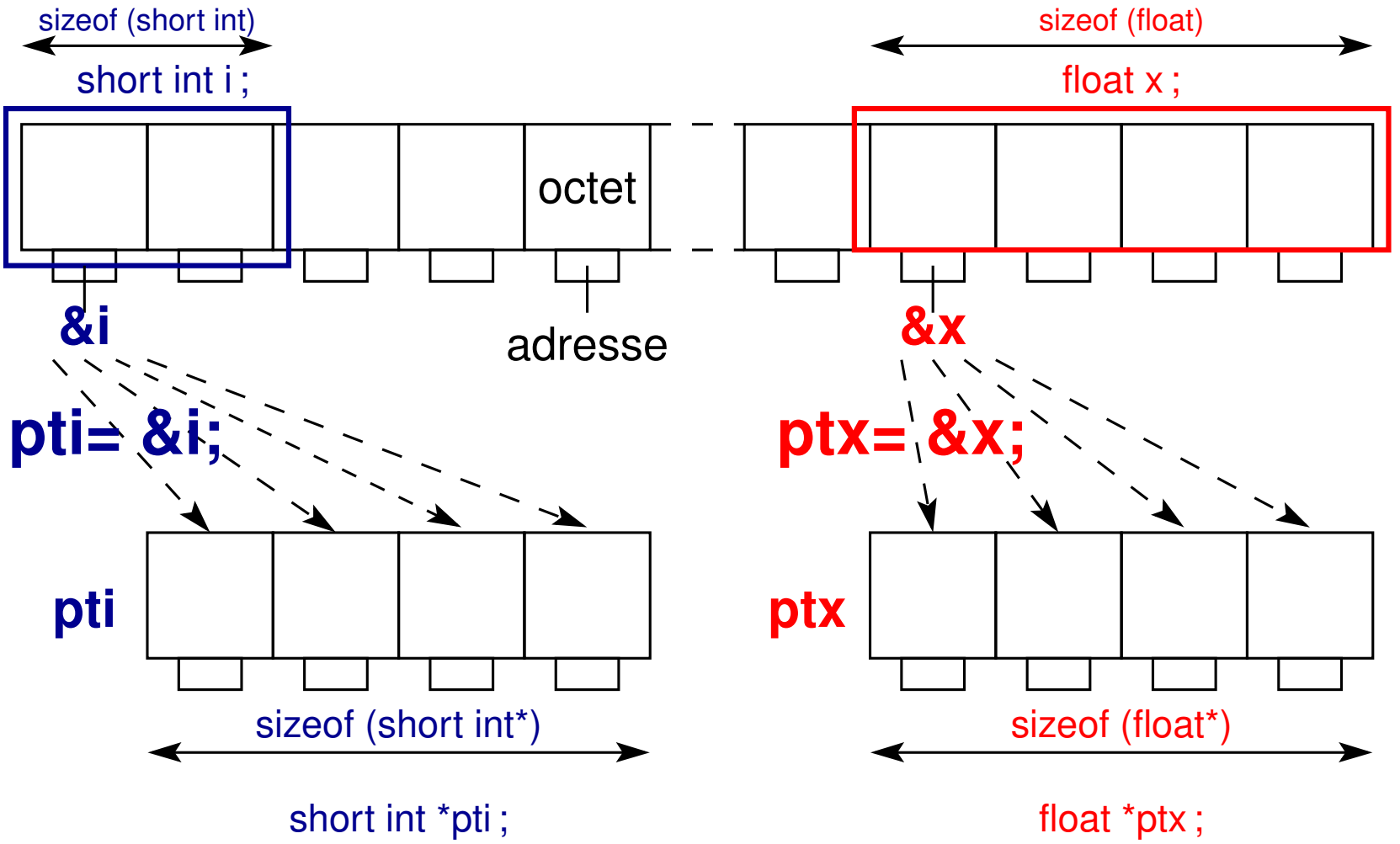
Affecter la **valeur d'un pointeur** `pty` **à un pointeur** de même type `ptx` :

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

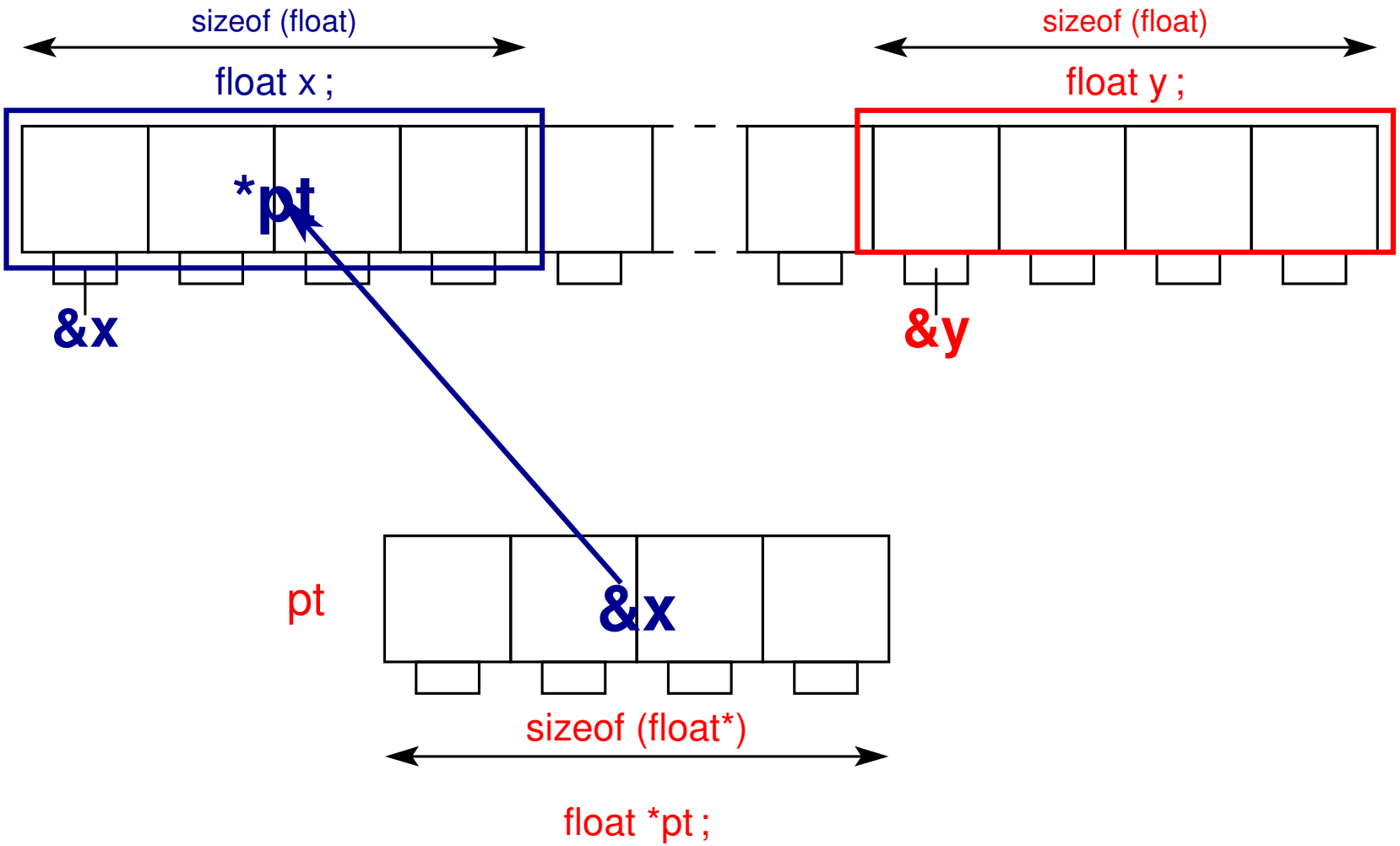
Faire pointer un pointeur vers une variable =

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

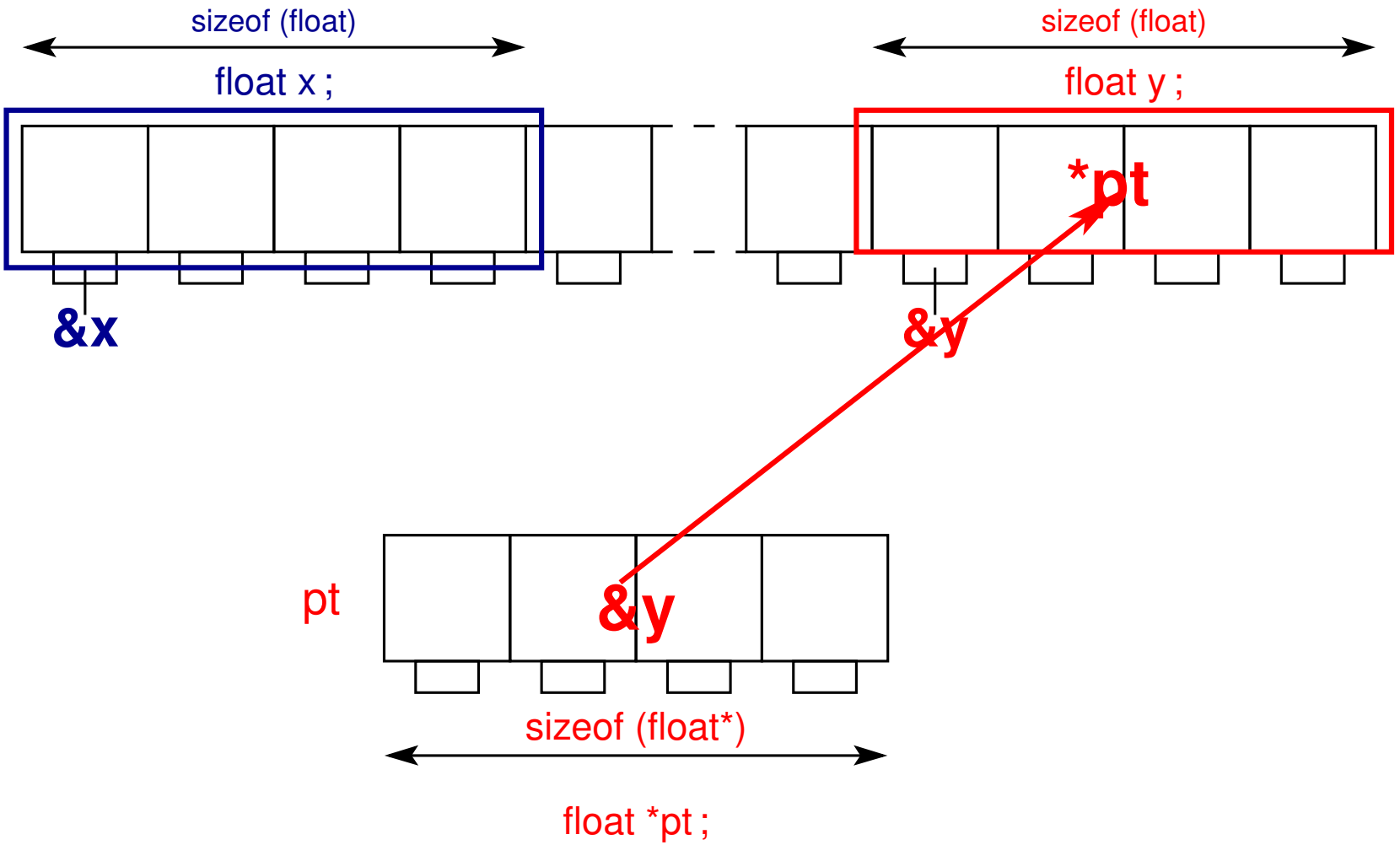
= affecter une valeur à la variable pointeur



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` vaut désormais `4`

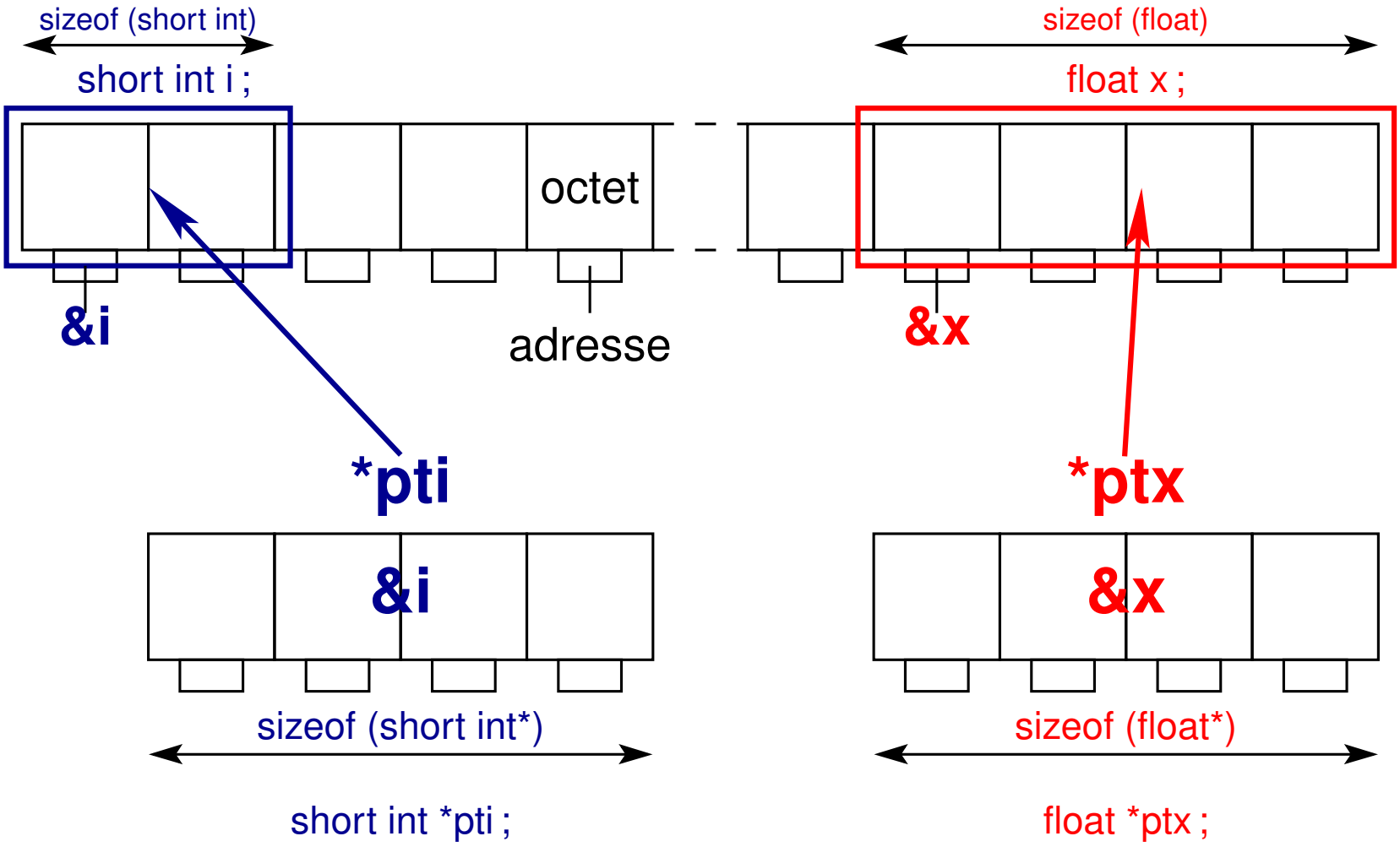
Attention :

— pour que le codage/décodage soit correct, il faut que le pointeur soit un pointeur vers une variable de **même type que la cible**.

— affectation de la cible désastreuse si le pointeur n'a pas été initialisé
⇒ modification possible d'une autre variable (erreur aléatoire sournoise)
ou accès à une zone mémoire interdite (**`segmentation fault`** : mieux !)

Opération d'**indirection** (*) sur un pointeur =

accès 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	`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)

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

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

6.6 Initialiser les pointeurs !

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

L'adresse qu'il contient est aléatoire

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

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

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

```
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\npj=%019lu\n", (unsigned long int) pi,
                                 (unsigned long int) pj);
i = 1;
/* suivant l'ordre de decl. *pi, *pj => ?  mais *pj, *pi erreur */
printf("i=%d, *pi=%d , *pj=%d\n", i, *pi, *pj); /* aleatoire*/
/* si l'affichage se fait, il est aleatoire (statique/dynamique) */
⟹*pj = 2; /* => erreur fatale en écriture */
/* accès à une zone interdite quand on affecte 2 à l'adresse pj */
```

```
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
```