

UPMC

Master P&amp;A/SDUEE

UE MNI (4P009)

Méthodes Numériques et Informatiques

<b>Fortran 95/2003 et C</b>
-----------------------------

Sofian.Teber@lpthe.jussieu.fr

Jacques.Lefrere@upmc.fr

2015–2016

Albert Hertzog

Fortran et C

<b>2 Types et déclarations des variables</b>	<b>19</b>
2.1 Représentation des nombres : domaine ( <i>range</i> ) et précision	19
2.1.1 Domaine des entiers signés	19
2.1.2 Limites des entiers sur 32 et 64 bits	21
2.1.3 Domaine et précision des flottants	22
2.1.4 Caractéristiques numériques des flottants sur 32 et 64 bits	26
2.1.5 Caractéristiques des types numériques en fortran	27
2.2 Types de base	28
2.3 Les constantes	30
2.4 Déclarations des variables	31
<b>3 Opérateurs</b>	<b>33</b>
3.1 Opérateur d'affectation	33

## Table des matières

<b>1 Introduction</b>	<b>6</b>
1.1 Langage compilé et langage interprété	6
1.2 Programmation en langage compilé	6
1.3 Compilation et édition de liens	8
1.4 Historique	12
1.4.1 Langage fortran	12
1.4.2 Langage C	13
1.5 Intérêts respectifs du C et du fortran	14
1.6 Format des instructions	16
1.7 Exemple de programme C avec une seule fonction utilisateur	17
1.8 Exemple de programme fortran avec une seule procédure	18
3.2 Opérateurs algébriques	35
3.3 Opérateurs de comparaison	35
3.4 Opérateurs logiques	36
3.5 Incrémentation et décrémentation en C	38
3.6 Opérateurs d'affectation composée en C	39
3.7 Opérateur d'alternative en C	39
3.8 Opérateur sizeof en C	40
3.9 Opérateur séquentiel «,» en C	40
3.10 Opérateurs & et * en C	40
3.11 Priorités des opérateurs en C	41
<b>4 Entrées et sorties standard élémentaires</b>	<b>42</b>
4.1 Introduction aux formats d'entrée–sortie	43

MNI

1

2015-2016

Fortran et C

4.1.1 Introduction aux formats en C . . . . . 46

**5 Structures de contrôle 48**

5.1 Structure conditionnelle *if* . . . . . 49

5.1.1 Condition *if* . . . . . 49

5.1.2 Alternative *if ... else* . . . . . 49

5.1.3 Exemples d'alternative *if ... else* . . . . . 51

5.1.4 Alternatives imbriquées *if ... else* . . . . . 53

5.1.5 Aplatissement de l'imbrication avec *else if* en fortran . . . . . 54

5.2 Aiguillage avec *switch/case* . . . . . 55

5.2.1 Exemples d'aiguillage *case* . . . . . 56

5.3 Structures itératives ou boucles . . . . . 60

5.3.1 Exemples de boucle *for* ou *do* . . . . . 62

5.4 Branchements ou sauts . . . . . 64

5.4.1 Exemples de bouclage anticipé *cycle/continue* . . . . . 66

5.4.2 Exemples de sortie anticipée de boucle via *break/exit* . . . . . 67

**6 Introduction aux pointeurs 68**

6.1 Intérêt des pointeurs . . . . . 68

6.2 Pointeurs et variables : exemple du C . . . . . 69

6.2.1 Affectation d'un pointeur en C . . . . . 72

6.2.2 Indirection (opérateur \* en C) . . . . . 76

6.3 Pointeurs en fortran . . . . . 78

6.4 Syntaxe des pointeurs (C et fortran) . . . . . 79

6.5 Exemples élémentaires (C et fortran) . . . . . 80

6.6 Initialiser les pointeurs ! . . . . . 82

# 1 Introduction

## 1.1 Langage compilé et langage interprété

Langage compilé	Langage interprété
<b>C</b> , C++, <b>fortran</b>	<b>shell</b> , <b>python</b> , perl, php, scilab
<b>Le compilateur</b> analyse l'ensemble du programme avant de le traduire en langage machine une fois pour toutes. ⇒ <b>optimisation</b> possible L'exécutable produit est <b>autonome</b> mais dépend du processeur	<b>L'interpréteur</b> exécute le code source ( <b>script</b> ) instruction par instruction. ⇒ 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
<b>Compilé + interprété</b> , par exemple <b>java</b> compilation → 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 :

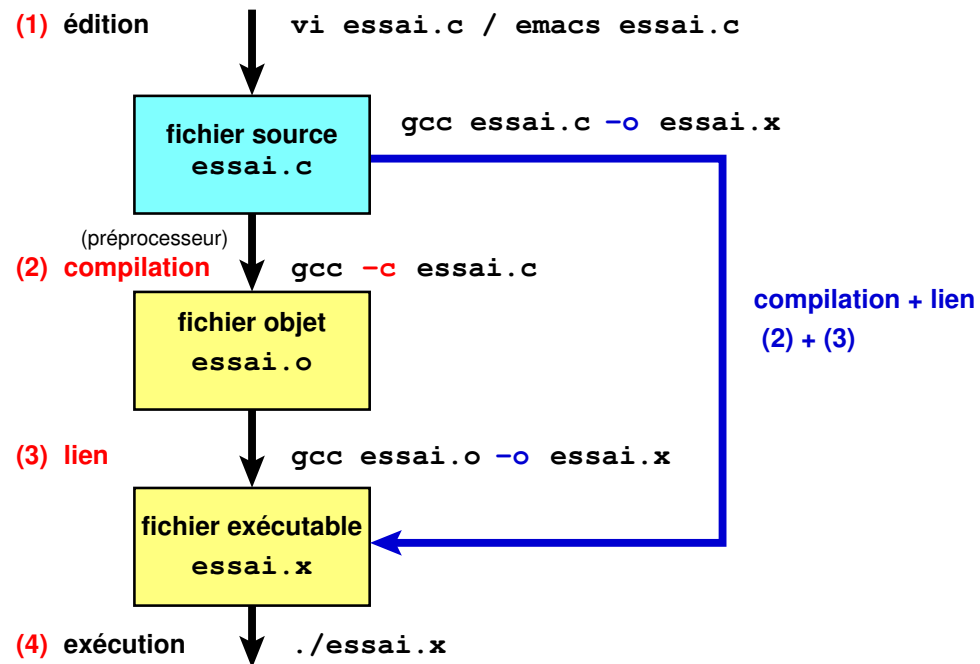
1. traitement par le **préprocesseur (cpp)** des lignes commençant par **#** appelées directives (transformation textuelle)
2. **compilation** à proprement parler → fichier objet `essai.o`
3. **édition de liens (link)** : le compilateur lance `ld` → fichier exécutable `a.out` assemblage des codes objets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

#### 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	<b>gcc</b>	<b>gfortran</b>
Documentation	<a href="http://gcc.gnu.org">http://gcc.gnu.org</a>	
Intel	<code>icc</code>	<code>ifort</code>



#### 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	
<code>gcc-mni-c89</code>	<code>gfortran-mni</code>
<code>gcc-mni-c99</code>	<code>gfortran2003-mni</code>

## 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...  
⇒ **ne plus écrire de fortran 77**
- 1997 : **fortran 95** = mise à jour mineure
- 2004 : **fortran 2003** standard adopté  
nouveau : **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

## 1.5 Intérêts respectifs du C et du fortran

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

### 1.4.2 Langage C

- langage conçu dans les années 1970
- 1978 : **The C Programming Language** de B. KERNIGHAN et D. RICHIE
- développement lié à la diffusion du système UNIX
- 1988–90 : normalisation **C89 ANSI-ISO** (bibliothèque standard du C)  
Deuxième édition du KERNIGHAN et RICHIE **norme ANSI**
- 1999 : norme **C99**  
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, ...

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
<b>libre</b>	format du code	<b>libre</b> 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 <b>&amp;</b> à la fin
entre /* et */ peut s'étendre sur plusieurs lignes	délimiteurs de commentaire	
// en C99 et C++	introduceur de commentaire → fin de la ligne	<b>!</b>
<b>distinction</b>	maj/minuscule	pas de distinction
lignes de directives pour le préprocesseur : <b>#</b> en première colonne		

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

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

## 1.8 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) ⇒ **domaine couvert limité**

Mais distinguer :

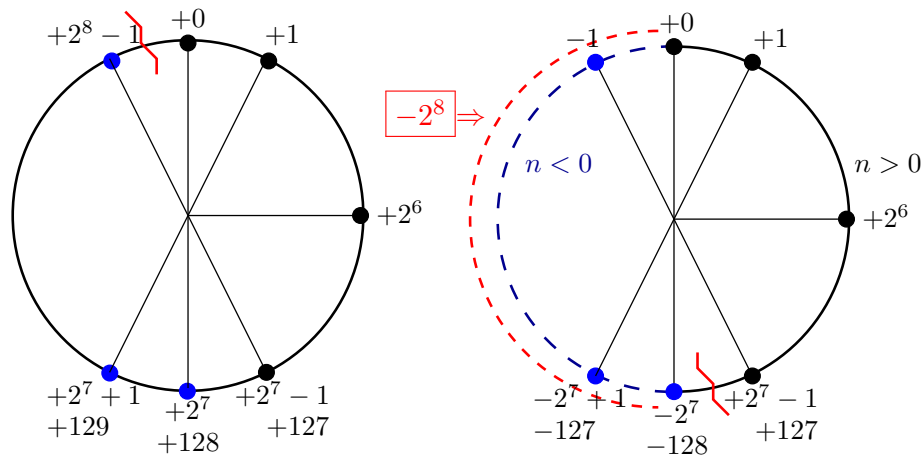
- les **entiers représentés exactement**
- les **réels représentés approximativement** en virgule flottante

⚠ ⇒ 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	$0 \rightarrow 255 = 2^8 - 1$	$-2^7 = -128 \rightarrow +127 = 2^7 - 1$



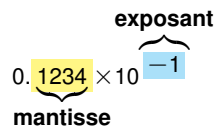
Entiers positifs sur 8 bits  
 Discontinuité en haut à gauche de 0  
 Entiers relatifs sur 8 bits  
 Discontinuité en bas entre +127 et -128  
 Pour passer des positifs aux relatifs, on soustrait  $2^8$  dans la partie gauche du cercle.

2.1.3 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	$0.1234 \times 10^{-4}$
0.000123456789	.0001	$0.1234 \times 10^{-3}$
0.00123456789	.0012	$0.1234 \times 10^{-2}$
0.0123456789	.0123	$0.1234 \times 10^{-1}$
0.123456789	.1234	$0.1234 \times 10^0$
1.23456789	1.2345	$0.1234 \times 10^1$
12.3456789	12.3456	$0.1234 \times 10^2$
123.456789	123.4567	$0.1234 \times 10^3$

Virgule flottante en base 10



2.1.2 Limites des entiers sur 32 et 64 bits

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

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

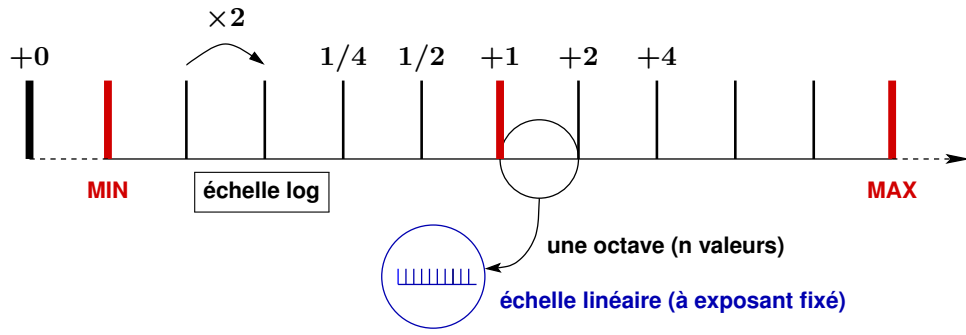
- ⚠ Dépassement de capacité en entier positif  $\Rightarrow$  passage en négatif
- $\rightarrow$  en fortran, choix des variantes (KIND) d'entiers selon le domaine (range) par la fonction **SELECTED\_INT\_KIND(...)**
- $\rightarrow$  en C89, les tailles des entiers dépendent du processeur
- $\rightarrow$  C99 : types entiers étendus à nb d'octets imposé, par exemple : **int32\_t**

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
- $\epsilon =$  la plus petite valeur telle que  $1 + \epsilon$  soit le successeur de 1
- $\epsilon$  est le pas des flottants dans l'octave  $[1, 2[ \Rightarrow \epsilon = 1/2^m$
- Précision relative  $\leq \epsilon = 1/2^m$
- Flottants sur 32 bits :  $m = 23$  bits de mantisse  $\Rightarrow \epsilon = 2^{-23} \approx 10^{-6}/8$
- 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** )

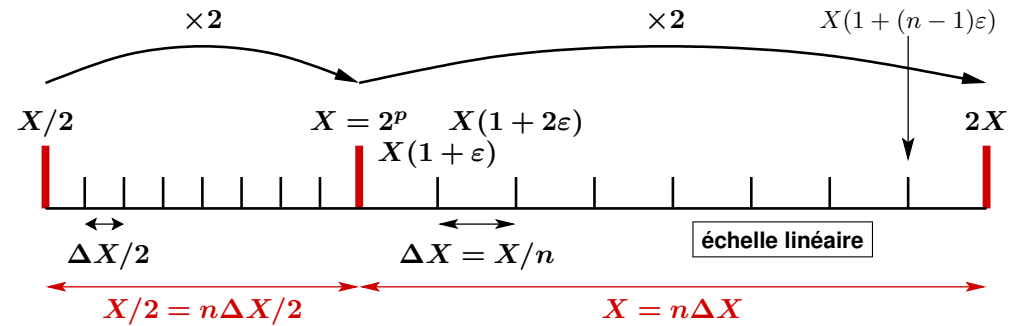
**2.1.4 Caractéristiques numériques des flottants sur 32 et 64 bits**

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

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

**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  $\Delta X = X/n = \epsilon X$



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

**2.1.5 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  
 $k_i =$  **SELECTED\_INT\_KIND** (r) pour les entiers allant jusqu'à  $10^r$   
 $k_r =$  **SELECTED\_REAL\_KIND** (p, r) pour les réels  
 pour un domaine de  $10^{-r}$  à  $10^r$  et une précision  $10^{-p}$  (p chiffres significatifs)

## 2.2 Types de base

### Types représentés **exactement**

langage C	Type	fortran 90
<code>////C99 bool</code>	booléen	<b>logical</b>
<code>char</code>	caractère	$\approx$ <code>character(len=1)</code>
<code>////(tableau de char)</code>	chaîne de caractères	<b>character(len=....)</b>
<code>short int</code>	entier court	<code>integer(kind=2)</code>
<b>int</b>	entier par défaut	<b>integer</b>
<code>long int</code>	entier long	<code>integer(kind=4/8)</code>
<b>C99 long long</b>	entier long long	<code>integer(kind=8)</code>

## 2.3 Les constantes

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



### Types représentés **approximativement**

en virgule flottante : mantisse et exposant

langage C	Type	fortran 90
<code>float</code>	réel simple précision (32 bits)	<b>real</b>
<b>double</b>	double précision (64 bits)	<code>double precision</code>
<code>long double</code>	précision étendue ( $\geq 80$ )	<code>real(kind=10/16)</code>
<code>////C99 complex</code> <code>#include &lt;tgmath&gt;</code>	complexe + variantes	<b>complex</b>

## 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 $\geq 89$	fortran $\geq 90$
en tête des blocs <b>C99 N'importe où</b>	<b>en tête des procédures</b>
Syntaxe	
type identificateur1, identificateur2 ... ;	type :: identificateur1, identificateur2, ...
Exemple : déclaration de 3 entiers	
<b>int</b> i, j2, k_max ;	<b>integer</b> :: i, j2, k_max
Initialisation : lors de la déclaration	
<b>int</b> i = 2 ; (exécution)	<b>integer</b> :: i = 2 (compilation)
Déclaration de constantes (non modifiables)	
<b>const int</b> i = 2 ; penser aussi à #define VAR 2	<b>integer, parameter</b> :: 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
<b>lvalue = (type) expression</b>	<b>variable = type (expression)</b>
⇒ <b>conversion forcée</b> (opérateur <b>cast</b> )	grâce à des fonctions intrinsèques
entier = <b>(int)</b> flottant;	entier = <b>INT</b> (flottant)

```

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	≈ <b>pow(x, y)</b>	<b>**</b>	
reste modulo	<b>%</b>	≈ <b>mod(i, j)</b>	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 <sup>a</sup>	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 ≠ 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` ⇒ `lvalue = lvalue opérateur expression`**

Exemples :

<code>j += i</code>	⇒	<code>j = j + i</code>
⚠ <code>b *= a + c</code>	⇒	<code>b = b * (a + c)</code>

### 3.7 Opérateur d'alternative en C

**`exp1 ? exp2 : exp3` ⇒ 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          sizeof (type)
size_t n1; double a;          size_t n2;
n1 = sizeof a;                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** ⇒ adresse de l'objet

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

## 4 Entrées et sorties standard élémentaires

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

### 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 `,`
- ⇒ indiquer les priorités avec des parenthèses !

### 4.1 Introduction aux formats d'entrée-sortie

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

	c	fortran 2003
<b>entiers</b>		
décimal	<code>%w[.p]d</code>	<code>Iw[.p]</code>
	<code>%d</code>	<code>I0</code>
<b>réels</b>		
virgule fixe	<code>%w[.p]f</code>	<code>Fw.p</code>
virgule flottante	<code>%w[.p]e</code>	<code>Ew.p</code>
préférer ⇒ général	<code>%w[.p]g</code>	<code>Gw.p</code>
<b>caractères</b>		
caractères	<code>%w[.p]c</code>	<code>A[w]</code>
chaîne	<code>%w[.p]s</code>	<code>A[w]</code>

```
#include <stdio.h> /* entrées sorties standard */
#include <stdlib.h>
int main(void) {
int i;
float x;
double y;
printf("Entrer un entier\n");
scanf("%d", &i); /* passer l'adresse */
printf("La valeur de i est %d\n", i);
printf("Entrer un float, un double \n");
scanf("%g %lg", &x, &y); /* passer les adresses */
printf("x = %g et y = %g\n", x, y);
exit(EXIT_SUCCESS);
}
```

#### 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	<b>%c</b>
chaîne	<b>%s</b>
short (converti en)/ <b>int</b>	<b>%d</b>
long	%ld
long long	%lld
<b>float</b> (convertis en)/ <b>double</b>	(%e, %f) <b>%g</b>
long double	(%Le, %Lf) %Lg



```
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
```

##### En entrée avec scanf

Type	Format
char	<b>%c</b>
short	%hd
<b>int</b>	<b>%d</b>
long	%ld
long long	%lld
<b>float</b>	(%e, %f) <b>%g</b>
<b>double</b>	(%le, %lf) <b>%lg</b>
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.

⇒ trop restrictif

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

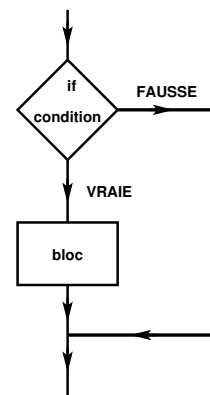
- exécution **conditionnelle** (**if / else**) ou **aiguillage** (**case**) dans les instructions
- **itération** de certains blocs (**for, do, while...**)
- **branchements** (**cycle** ou **continue, exit** ou **break, ...**)

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

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

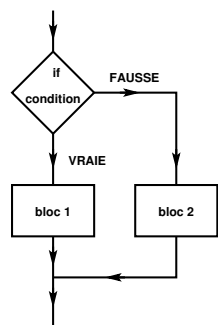
### 5.1 Structure conditionnelle if

#### 5.1.1 Condition if



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

#### 5.1.2 Alternative if ... else



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

#### 5.1.3 Exemples d'alternative if ... else

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

5.1.5 Aplatissement de l'imbrication avec else if en fortran

Structures imbriquées => deux **end if**

Structure aplatie => un **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
    
```

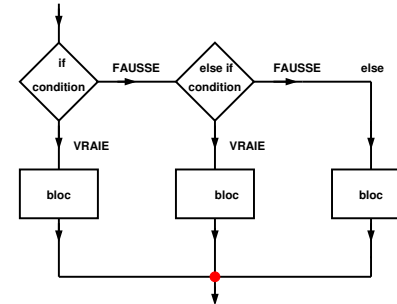
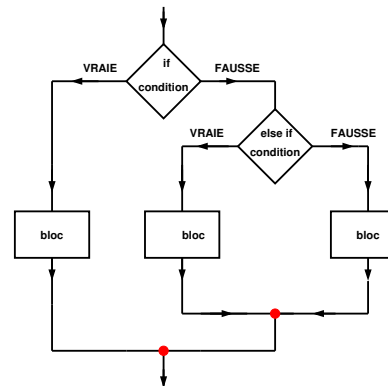
```

! structure 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
    
```

5.1.4 Alternatives imbriquées if ... else

**Imbrication simple** => deux niveaux

**Fusion des retours** => un niveau



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

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

## 5.2.1 Exemples d'aiguillage case

```

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

```

```

_____ 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 _____
/* 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") ;
}

```

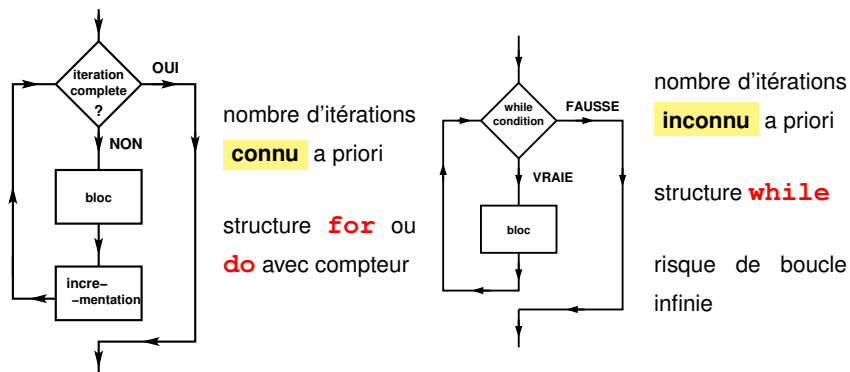
```

_____ 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 :



### 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) ;

```

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

```

_____ 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

```

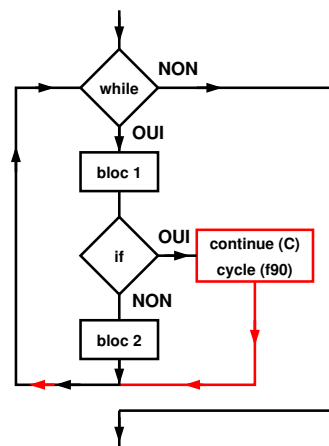


## 5.4 Branchements ou sauts

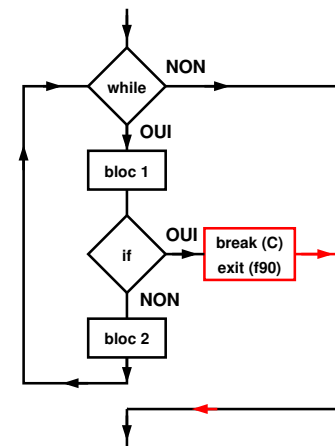
C		fortran 90
<code>continue ;</code>	bouclage anticipé	<code>cycle</code>
<code>break ;</code>	sortie anticipée	<code>exit</code>
<code>goto étiquette ;</code> <sup>a</sup>	branchement (à éviter)	<code>go to étiquette-numérique</code>

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`

```
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`

```
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

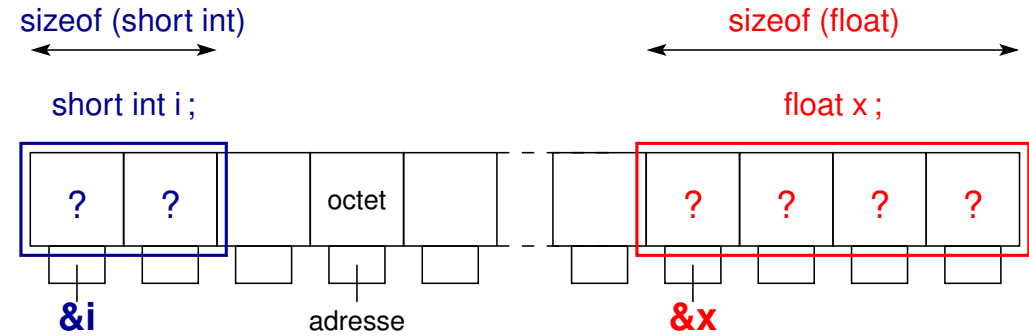
### 6.1 Intérêt des pointeurs

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

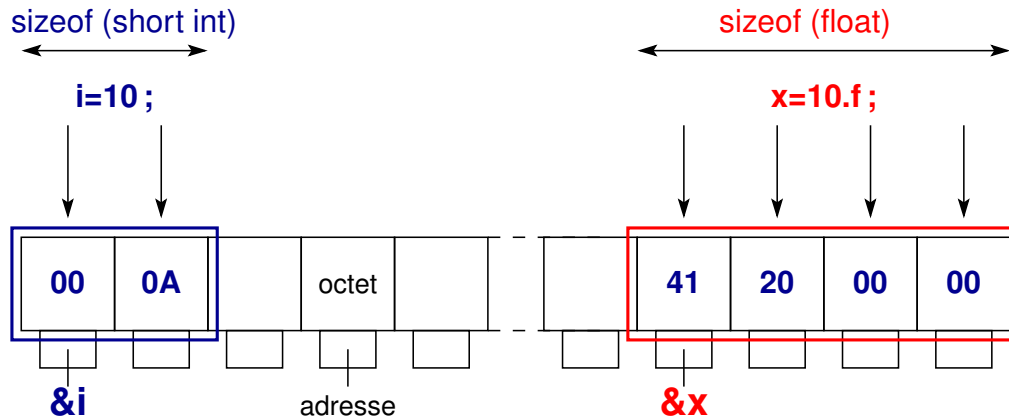
<b>indispensables en C</b>	absents en fortran 77,
pour les arguments des fonctions	mais introduits en fortran 90/95
et les tableaux dynamiques	et étendus en fortran 2003
stockent des <b>adresses</b> des variables	pas d'accès aux adresses
- **utilisation commune** : tris sur des données volumineuses,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

### 6.2 Pointeurs et variables : exemple du C

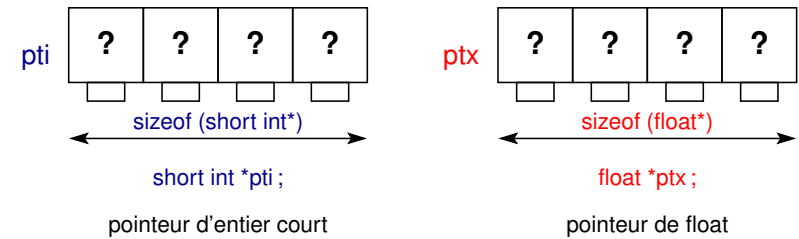
Déclarer une variable d'un **type** donné = réserver une zone mémoire dont la **taille** (`sizeof` en C) dépend du type et le **codage** est fixé par le type



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** ⇒ leur **type** indique la **taille** et le **codage** de la **cible** potentielle



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

### 6.2.1 Affectation d'un pointeur en C

Affecter l'adresse d'une variable à un pointeur :

```
pti = &i; ptx = &x;
```

= copier l'adresse mémoire de la variable cible (opérateur &)  
dans la zone mémoire réservée lors de la déclaration du pointeur.

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

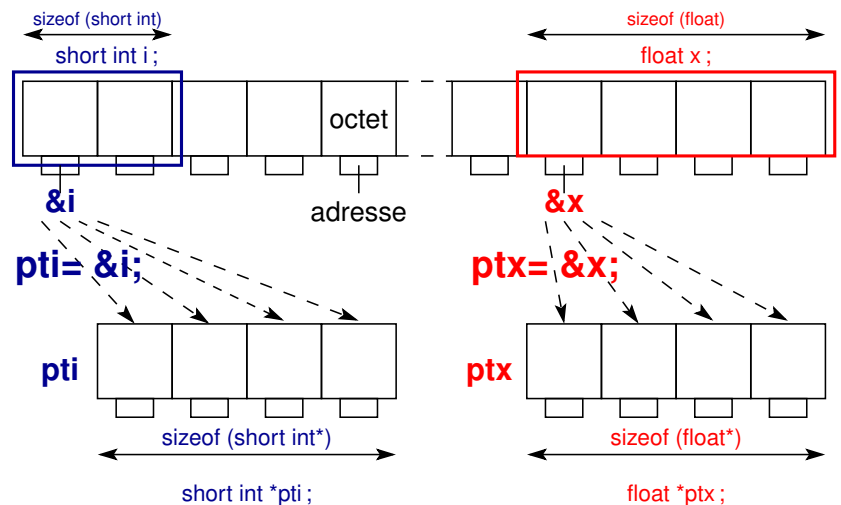
#### Attention :

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

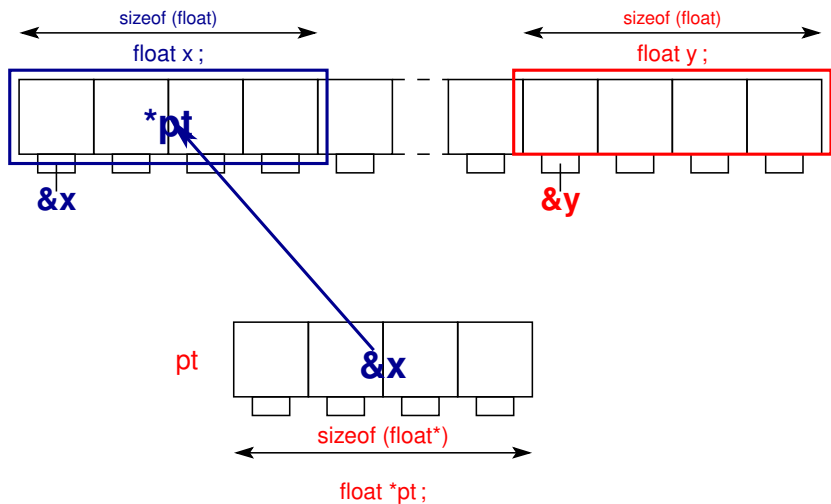
Affecter la valeur d'un pointeur **pty** à un pointeur de même type **ptx** :

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

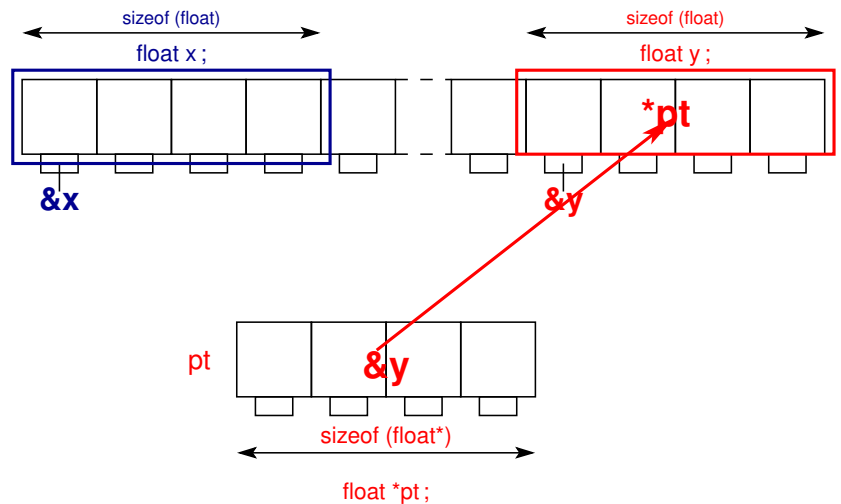
Faire pointer un pointeur vers une variable =  
écrire l'adresse de la variable dans les cases réservées pour le pointeur  
= affecter une valeur à la variable pointeur



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).

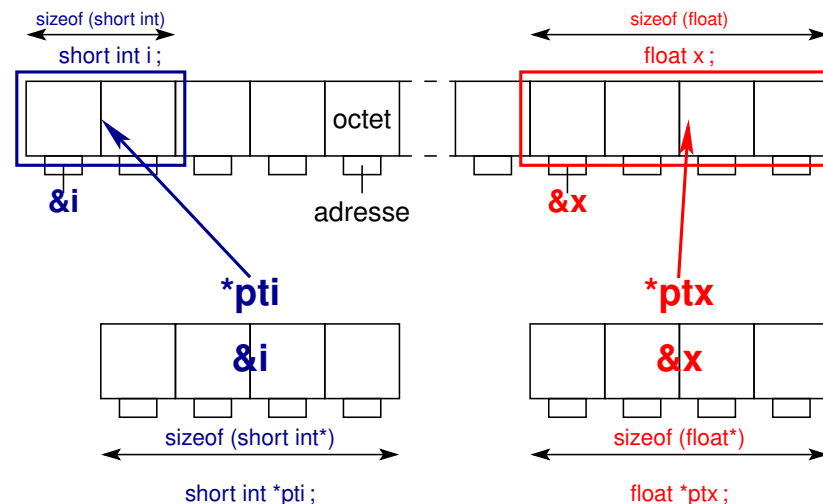
`j = *pti;` la variable `j` prend la valeur de la cible de `pti`

`*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  
 ⇒ 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
<code>type *ptr;</code>	déclaration	<code>type, pointer :: ptr</code>
<code>type *ptr=NULL;</code>	avec initialisation	<code>type, pointer :: ptr=&gt;null()</code>
<code>type var;</code> (pas d'attribut)	cible	<code>type, target (nécessaire) :: var</code>
<code>ptr = &amp;var ;</code>	pointer sur	<code>ptr =&gt; var</code> (associer <code>ptr</code> à <code>var</code> )
<code>*ptr</code>	variable pointée par	<code>ptr</code>
<code>ptr = NULL ;</code>	dissocier	<code>nullify(ptr); ptr=&gt;null()</code>
	associé ? à la cible <code>var</code> ?	<code>associated(ptr)</code> <code>associated(ptr, var)</code>
⚠	concerne les <b>adresses</b> !	<code>ptr2 = ptr1</code> concerne les <b>cibles</b> !

## 6.5 Exemples élémentaires (C et fortran)

```

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

```

MNI

80

2015-2016

6 Introduction aux pointeurs

Fortran et C

6.6 Initialiser les pointeurs !

## 6.6 Initialiser les pointeurs !

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

L'adresse qu'il contient est aléatoire

⇒ le résultat d'une indirection est aussi aléatoire

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

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

```

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

```

MNI

81

2015-2016

6 Introduction aux pointeurs

Fortran et C

6.6 Initialiser les pointeurs !

```

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 */

```

MNI

82

2015-2016

MNI

83

2015-2016

```

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

```