La « vie » sans Notion de Type

Au début du développement d'Unix, que ce soit en langage B ou en langage BCPL, la notion de type n'existait pas :

- La programmation se faisait en « cellule » ou « mot » mémoire.
- L'évolution rapide du matériel a très vite démontré que **cette approche n'était pas viable** !

Un flottant, d'une machine à l'autre, n'occupait pas forcément le même nombre de cellules.

- Il fallait ré-écrire le code pour chaque machine.

La solution :

Abstraire la notion de flottant au niveau du langage de programmation

- **On écrirait que la variable est flottante** . . .
- C'est **le compilateur qui gèrerait la mise en correspondance avec le nombre de cellules mémoire** nécessaires !

Notion de Type

Type :

Un **type de données** définit une **collection d'objets et un ensemble d'opérations** qui agissent sur ces objets.

L'intérêt du typage est double :

① Il permet de préciser et de mieux définir le modèle de données : c'est donc un outil d'expression.

```
float x;
char c;
```

② Il permet aux phases de compilation, et donc au compilateur de procéder à des contrôles de « cohérence » qui permettent d'éviter des erreurs de modélisation.

```
extern  double pow(double X, double Y);
```

Typage statique

Le C est un langage **typé statiquement** :

- ➢ chaque variable, chaque constante, chaque fonction, chaque expression . . . a un **type défini à la compilation**.

Cette démarche présente deux avantages :

1. ① vérification,
2. ② et performance.

Mais elle est bride l'expressivité d'un code informatique :

- ➢ on ne connaît pas toujours statiquement la nature exacte des choses que l'on souhaite manipuler.

  Par exemple, lorsque on accède à un fichier disque, il faut attendre d'avoir lu quelques éléments pour savoir le type exact de l'information qu'il contient : image noir/blanc ou couleur . . .

Conversion et Débrayage de type

Il est donc impossible de faire évoluer le type d'un objet (alloué en mémoire) en C.

Par contre :

- On peut convertir/transformer le type de la valeur dans le cadre d'une utilisation dans une expression.

  C'est la notion de coercition/cast.

  - Un réel peut par exemple être converti en un entier.
    Cette conversion n'est pas sans effet sur la valeur initiale. Dans ce cas, la partie décimale est tronquée.

- On peut aussi utiliser la notion de pointeur générique : (void *)

  On dit juste que la variable est une "référence à un objet" . . . **sans préciser** la nature exacte de l'objet.

  C'est approche est valide tant qu'on ne manipule que la référence.

  Dès qu'on va vouloir connaître sa valeur, il faudra avoir l'information du type de l'objet référencé et on utilisera alors la coercition pour préciser cela (ou les conversions implicites) !

Types de base scalaires

C met à disposition un ensemble de type primitifs **scalaires** :

① caractère : **char** (1 octet)
② entiers : **int**, **short**, **long**, . . .
③ réels : **float** ou **double**
④ énumération : **enum**
⑤ void : **void**

➢ On ne peut pas diviser/fractionner un scalaire,
➢ Un scalaire est donc un tout (non sécable) : ses manipulations (R/W) portent sur sa globalité.

Cela diffère des tableaux (agrégats homogènes) par exemple.

Au niveau de la mémoire, les scalaires occupent quelques "cases mémoires" avec un codage adéquat à ce qu'ils représentent.

Les caractères : **char**

Le type dont l'ensemble des valeurs est l'ensemble des valeurs entières formant le code des caractères (sur 1 octet) utilisé sur la machine cible.

- ➢ correspond à la plus petite taille adressable sur une machine.
- ➢ est considéré comme un sous-ensemble du type entier.
  a) Classiquement on l'utilise pour représenter des codes ASCII.
  ```
  char c = 'A';
  ```
  b) Il permet aussi de **réaliser des opérations arithmétiques sur des entiers plus « petits » que les entiers** . . . utile pour l'image par exemple.
- ➢ admet les qualificatifs **unsigned** ou **signed** .

char	1 Octet	$[-128, +127]$
unsigned char	1 Octet	$[0, 255]$

Encodage des caractères

Le code des caractères est un choix d'implémentation du compilateur.

- Code ASCII ou, de plus en plus souvent, le code ISO-8859-1 (Latin 1) qui permet de représenter des caractères accentués européens (ASCII étendu).

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

Source : www.LookupTables.com

Opérations sur caractères

Via le `man` et <ctypes.h> vous trouverez des fonctions utiles :

int isalpha(int c) : returns non-zero if c is a letter ('A'-'Z' or 'a'-'z').
int isupper(int c) : returns non-zero if c is a upper case letter (A-Z).
int islower(int c) : returns non-zero if c is a lower case letter ('a'-'z').
int isdigit(int c) : returns non-zero if c is a decimal digit ('0'-'9').
int isxdigit(int c) : returns non-zero if c is a hexadecimal digit.
int isspace(int c) : etc . . .
int ispunct(int c) :
int isalnum(int c) :
int isprint(int c) :
int isgraph(int c) :
int iscntrl(int c) :
int toupper(int c) :
int tolower(int c) :
int isascii(int c) :
int toascii(int c) :

Les caractères : **wchar_t**

Le standard ISO/ANSI C contient, dans une correction qui fut ajoutée en 1995, un type de caractère codé sur 16 bits : **wchar_t**.

Il y a aussi

- un ensemble de fonctions comme celles contenues dans `<string.h>` et `<ctype.h>` déclarées respectivement dans `<wchar.h>` et `<wctype.h>`,
- et un ensemble de fonctions de conversion entre **char** $*$ et **wchar_t** $*$ (déclarées dans `<stdlib.h>`).

Comment utiliser Unicode ? :
`www.freenix.fr/unix/linux/HOWTO/Unicode-HOWTO-5.html`

Dans cette url, vous pourrez trouver quelques limites à la portabilité concernant ce type :

> « Une variable wchar_t peut être encodée en Unicode ou non. »

Processus de normalisation du C

Il y a plusieurs **normes** pour écrire du C :

< 1983 , c'est « le C de Kernighan & Ritchie » (K&R C).

En 1983 , l'institut national américain de normalisation (ANSI) forme un comité de normalisation du langage qui aboutit en 1989 à l'approbation de la norme dite **ANSI C ou C89** (formellement ANSI X3.159-1989).

En 1990 , cette norme est également adoptée par l'ISO (formellement ISO/IEC 9899 :1990).

ANSI C est une évolution du K&R C avec lequel il reste extrêmement compatible.

En 1999 , une nouvelle évolution du langage est normalisée par l'ISO : **C99**.

Cette norme introduit les types **booléens** et **complexes**. Elle permet aussi la définition de tableaux de taille dynamique en pile d'exécution . . .

La réalité du processus de normalisation

L'effort louable de normalisation ne cache pas certains problèmes qui peuvent freiner la portabilité du langage C.

- La normalisation n'est pas toujours assez précise compte tenu des possibilités d'expression et de manipulation du langage C qui sont comparables à celles de l'assembleur.

  Exemples : int, wchar_t, ...

- Les compilateurs (Gnu, Microsoft, Sun, Compaq, ...) n'ont pas les mêmes comportements devant les failles de spécification ou les innovations des normes.

  Ils n'offrent pas, non plus, les mêmes possibilités.

Ecrire portable en C

Même si le C est un langage très (trans)portable, on peut très facilement écrire des choses non portables.

Ecrire portable en C :

Il faut écrire du code avec le manuel de référence sous une main et le guide des usages de l'autre.

Il faut toujours vérifier,

- en lisant les manuels des compilateurs,
- en essayant sur le système cible,
- et en parcourant le Web,

que le comportement est conforme à ce qui est souhaité.

Il est certain que ce n'est pas la définition de la portabilité que l'on pouvait espérer :-((

Ecrire portable en C

Pour limiter **les risques liés au non respect de la norme ANSI** :

- Choisir et Lire la documentation du compilateur :

  `https://gcc.gnu.org/`

  `https://gcc.gnu.org/onlinedocs/gcc/Standards.html`

- Compiler avec les indispensables options :
  - **-ansi** : Permet de choisir le dialecte C89.
  - **-pedantic** : Toute utilisation du C non conforme strictement à la norme ANSI provoquera un warning (qu'on reconnaît facilement, car il commence par « warning : ANSI C forbids/disallows/... »).
- N'utiliser que des fonctions de bibliothèque ayant le label « ANSI ». C'est écrit dans le `man` !

Les nombres et les ordinateurs

Dans les mathématiques de tous les jours, nous utilisons un système positionnel qui permet, tout en utilisant un petit nombre de symboles, de représenter beaucoup de nombres :

$$\begin{array}{lll} x & \text{s'écrit} & s_n s_{n-1} \ldots s_1 s_0 \\ & = & s_n * b^n + s_{n-1} * b^{n-1} + \ldots + s_1 * b^1 + s_0 * b^0 \end{array}$$

- La valeur associée à chaque symbole dépend de sa position et de la base de travail (qui influe sur le nombre de symboles).
- Cette valeur vient contribuer par sommation à la valeur du nombre.

Par exemple, si on utilise la base 10 :

- $117 = 1 * 10^2 + 1 * 10^1 + 7 * 10^0$

ou encore

- $3.14 = 3 * 10^0 + 1 * 10^{-1} + 4 * 10^{-2}$

Dans le monde mathématique, l'écriture n'est pas limitée à un nombre maximum de symboles.

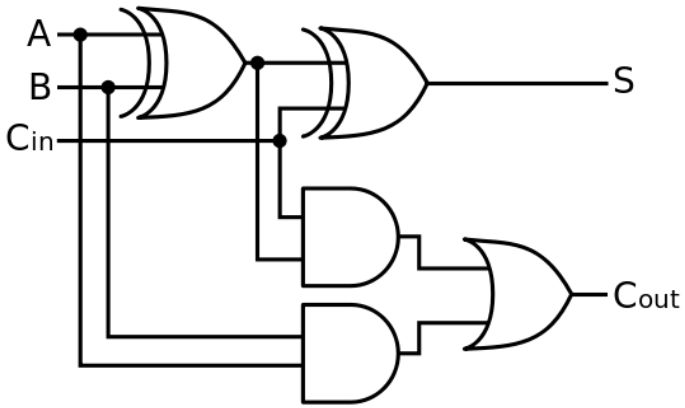
➢ On peut donc imaginer manipuler des valeurs et des précisions très grandes.

Les ordinateurs rencontrent plusieurs difficultés/problèmes relativement à l'abstraction des nombres. Ils doivent :

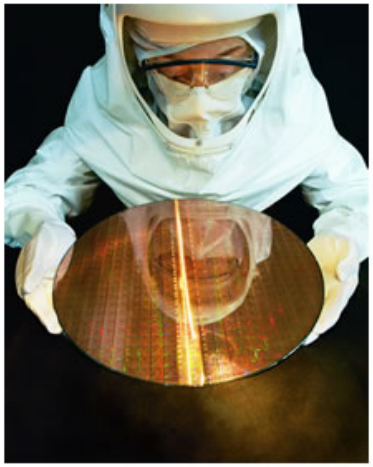
① Mémoriser ces nombres,

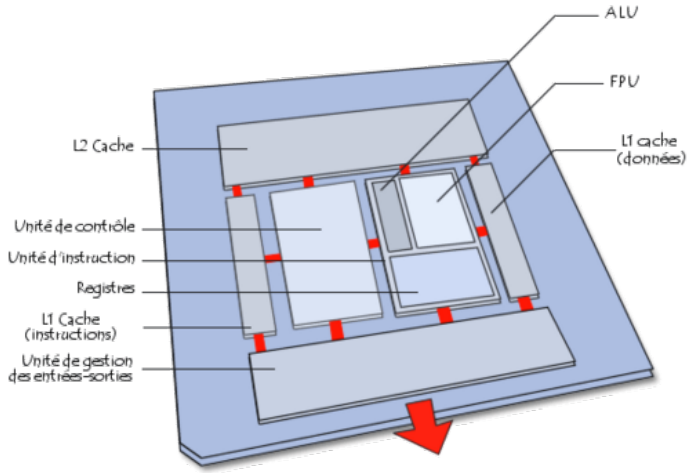
② Etre capable de les additionner, les soustraire, . . .

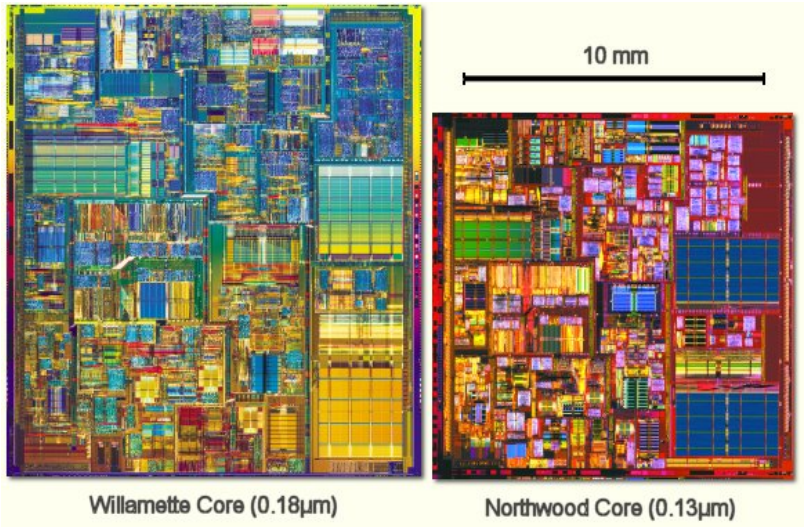
Dans les deux cas, des circuits logiques vont réaliser ces actions (ici additionneur complet sur 1 bit avec portes AND,OR et XOR).



Augmenter la taille des nombres, et donc des circuits logiques, influe sur la taille des composants, leur connectique, leur dissipation, leur prix . . .







Représentation binaire d'un nombre entier

Ces circuits logiques sont basés sur l'algèbre de Boole qui définit deux valeurs de vérité : vrai/faux que l'on encode par 0/1 (bit : Binary Information Unit).

- **Souvent pour se rapprocher de la réalité machine, on utilise la base 2 pour représenter les entiers :**

$$117_{10} = 1110101_2$$

Cette notation binaire permet de représenter exactement le même nombre que celui représenté en décimal.

Cette représentation permet de mieux cerner le nombre de symboles impliqués dans la manipulation du nombre.

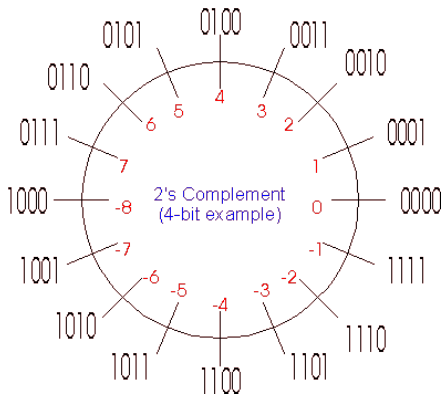
- Il ne faudrait pas que le nombre soit trop grand sinon les circuits matériels n'aurait pas les capacités pour manipuler ces nombres.
- Mais**si on reste dans la plage de valeurs autorisées par le nombre de symboles possibles dans la machine, on peut représenter sans approximation ni erreur le nombre entier.**

Représentation binaire et codage

Si la représentation binaire permet de représenter exactement un entier (idem représentation décimale) et veut approcher l'objet réellement manipulé par la machine, elle n'en reste pas moins "formelle".

La nécessité de représenter aussi les nombres entiers négatifs induit l'utilisation d'un codage. Plusieurs solutions sont possibles :

- ➡ Code binaire d'amplitude avec bit de signe,
- ➡ Code complément à 2 (Représentation unique du zéro !),



Les codages entiers ne modifient pas l'exactitude de la représentation (sauf dépassement de capacité).

Les entiers : **int**

En C, les **int** permettent de définir des variables de type "entier signé",

- ➢ très généralement en représentation "en complément à 2", bien que cela ne soit pas imposé par le langage.

Les entiers peuvent être affectés de deux types d'attributs :

① un attribut de précision : **short** ou **long** .

- ➢ Ils servent à qualifier un type scalaire de base pour **moduler la taille de l'occupation mémoire et le domaine de valeur induit**.

② un attribut de représentation : **unsigned** ou **signed**

- ➢ Les **unsigned int** permettent de contenir des entiers non signés en représentation binaire.
- ➢ L'attribut **signed** n'est que le pendant de **unsigned** .

Entiers : La portabilité

La taille des types n'est que **partiellement standardisée**

➢ Le standard C fixe uniquement une taille minimale et une magnitude minimale :

Entiers	Taille en Octets	Domaine de valeurs
int	2/4	$[-2147483648, +2147483647]$
short int ou **short**	2	$[-32768, +32767]$
long int	2/4/8	$[-2147483648, +2147483647]$
long	4	$[-2147483648, +2147483647]$
unsigned int	2/4	$[0, +65535]$
unsigned short int	2	$[0, +65535]$
unsigned long int	2/4/8	$[0, +4294967295]$

Spécification volontairement "floue"

Cette souplesse permet au langage d'être efficacement adapté à des processeurs très variés, mais elle complique la portabilité des programmes écrits en C.

Type	Taille
`char`, `unsigned char`, `signed char` (C89)	≥ 8 bits
`short` (identique à `signed short`), `unsigned short`	≥ 16 bits
`int` (identique à `signed int`), `unsigned int`	≥ 16 bits (taille d'un mot machine)
`long` (identique à `signed long`), `unsigned long`	≥ 32 bits
`long long` (identique à `signed long long`), `unsigned long long` (C99)	≥ 64 bits

- Les **int** sont "**théoriquement** implémentés" sur ce qui est un mot « naturel » de la machine.

  Mais c'est déja faux sur un Alpha (True64) . . .
- Les **short int** sont implémentés plus courts que les int, sinon comme des int.
- Les **long int** sont implémentés **si possible** plus grands que les int, sinon comme des int.

Ceci rend les précisions très dépendantes de la machine cible :

D'un point de vue du programmeur, il y un certain flou sur la taille "finale" d'un entier !

Ou est le problème ?

Imaginons que vous écriviez un programme manipulant des couleurs au format RGB.

- Vous avez besoin de 8 bits par composantes, soit 24 bits pour une couleur.
- Si vous écrivez qu'une couleur est implémentée par un "int" (en pensant ainsi disposer de 32 bits), il se pourrait bien que sur certaines (petites) machines (montres, mobiles, . . .), **vous ne disposiez QUE DE 16 bits** ! ?

Conséquence :

- Beaucoup de programmeurs définissent leurs propres types **Intxy** :

```
typedef short int int16;
typedef int int32;
typedef long int int64;
```

- D'un point de vue du programmeur, la taille de l'entier est alors déterministe . . . quitte à adapter ces définitions en fonction de la machine utilisée !

Occupation mémoire

En utilisant l'opérateur unaire `sizeof` :

```
1   /** Afficher l'occupation mémoire induit par le typage    */
2   #include <stdio.h>
3
4   int main(int argc, char *argv[]){
5     int i;        /* La taille depend de l'implantation */
6     short j;    /* Entier sur 16 bits */
7     long k;     /* Entier sur 64 bits */
8     float a;    /* Flottant simple precision */
9     double b;   /* Flottant double precision */
10
11    printf("i sur %zu octets \n", sizeof(i));
12    printf("j sur %zu octets \n", sizeof(j));
13    printf("k sur %zu octets \n", sizeof(k));
14    printf("a sur %zu octets \n", sizeof(a));
15    printf("b sur %zu octets \n", sizeof(b));
16  }
```

```
menez@vtr /Users/menez/Enseignements
$ gcc sizeofscal.c

menez@vtr /Users/menez/Enseignements
$ ./a.out
i sur 4 octets
j sur 2 octets
k sur 8 octets
a sur 4 octets
b sur 8 octets
```

Représentation mémoire

```
1   /** Afficher le codage induit par le typage    */
2   #include <stdio.h>
3   int main(void ){
4     int *p;
5
6     /* Définition de variables de types différents */
7     char c;
8     int i;
9     float j;
10    unsigned k;
11
12    /* Initialisation de ces variables */
13    c = '7';
14    i = -1;
15    j = 1.5;
16    k = 7;
17
18    /* Affichage de leur codage     */
19    printf("\n c vaut %c  ", c);
20    printf("son codage est : 0x%X\n",c);
21
22    printf("\n i vaut %d ", i);
23    printf("son codage est : 0x%X\n", i);
24
25    printf("\n j vaut %f ", j);
26    p = (int *) &j; /* Pas très joli ! */
27    printf("son codage est : 0x%X\n", *p);
28
29    printf("\n k vaut %d ", k);
30    printf("son codage est : 0x%X\n", k);
31
32    return 0;
33  }
34  /*-----------------------------------------*/
```

```
menez@vtr /Users/menez/EnseignementsCurrent/Co
$ gcc codage.c

menez@vtr /Users/menez/EnseignementsCurrent/Co
$ ./a.out

Ceci est un petit pgm de test des codes!

 c vaut 7  son codage est : 0x37

 i vaut -1 son codage est : 0xFFFFFFFF

 j vaut 1.500000 son codage est : 0x3FC00000

 k vaut 7 son codage est : 0x7
```

Les entiers : utilisation

La limitation de la magnitude d'un entier induite par la représentation que s'en font les machines peut avoir deux conséquences . . . **catastrophiques** :

- ➢ certaines initialisations perdront de leur sens et seront tronquées,
- ➢ certaines opérations peuvent engendrer un dépassement de capacité (**overflow**).

Troncature

```
1   /* Troncature */
2   #include <stdio.h>
3   int main(int argc, char *argv[]){
4     int i;        /* La taille dépend de l'implantation */
5     short j;    /* Entier sur 16 bits */
6     long k;
7     char c;
8
9     printf("1) Test de troncature !\n");
10
11    j = k = i= 72000;
12    printf("i = %d sur %ld octets donc 72000  \"in range\" \n", i, sizeof (i));
13    printf("k = %ld sur %ld octets donc 72000 \"in range\" \n", k,sizeof(k));
14    printf("j = %d sur %ld octets donc 72000 \"out range\" \n", j, sizeof (short));
15    printf("RMQ : j contient les %ld octets de poids faibles de k : %ld\n", sizeof(short), k&0x0FFFF );
16
17    return 0;
18  }
```

```
menez@vtr /Users/menez/EnseignementsCurrent/CodeC
$ gcc troncature.c

menez@vtr /Users/menez/EnseignementsCurrent/CodeC
$ ./a.out
1) Test de troncature !
i = 72000 sur 4 octets donc 72000  "in range"
k = 72000 sur 8 octets donc 72000 "in range"
j = 6464 sur 2 octets donc 72000 "out range"
RMQ : j contient les 2 octets de poids faibles de k : 6464
```

QUIZ : Si on affecte 72000 à la variable c, quelle sera la valeur de c ?

Dépassement de capacité

```c
/*  Dépassement */
#include <stdio.h>
int main(int argc, char *argv[]){
  int i;        /* La taille dépend de l'implantation */
  short s;
  unsigned short us;

  printf("1) Dépassement :\n");
  i = 0x7FFFFFFF; /* plus grande valeur positive sur 32 bits. */
  printf("Valeur de i \t: %d(%x) \n", i, i);
  i++;
  printf("Valeur de i+1 \t: %d(%x)\n", i, i);

  i = 0xFFFFFFFF; /* plus petite valeur negative sur 32 bits. */
  printf("Valeur de i \t: %d(%x) \n", i, i);
  i++;
  printf("Valeur de i+1 \t: %d(%x)\n", i, i);
}
```

```
menez@vtr /Users/menez/EnseignementsCurrent/CodeC
$ gcc depassement.c

menez@vtr /Users/menez/EnseignementsCurrent/CodeC
$ ./a.out
1) D◆passement :
Valeur de i     : 2147483647(7fffffff)
Valeur de i+1   : -2147483648(80000000)
Valeur de i     : -1(ffffffff)
Valeur de i+1   : 0(0)
```

```c
/*  Dépassement suite : sur signe / non signe */
#include <stdio.h>
int main(int argc, char *argv[]){
  int i;       /* La taille dépend de l'implantation */
  short s;
  unsigned short us;

  printf("\n2) Gestion des aspects signés ou non-signés :\n");
  i = 0xffffffff;
  printf("Un nombre : 0x%x mais deux Interpretations \n\t=> signée %d et non signé %u\n", i,i,i);

  s = us = 0x7FFF; /* plus grande valeur positive sur 16 bits. */
  printf("\nValeur de s : %d et de us : %d\n", s, us);

  /* Lors de l'appel à la fonction printf, ces valeurs sont converties en
     entiers longs avant d'être passées en paramètres : ceci donne lieu à
     une extension de signe. */
  printf("Meme représentation binaire de s : %x et de us : %x\n", s, us);
  s++; us++;
  printf("On incrémente l'évolution diffère (car les types diffèrent) !\n");
  printf("Valeur de s++ : %d et de us++ : %d\n", s, us);
  printf("Représentation de s : %x et de us : %x\n", s, us);
  s++; us++;
  printf("Valeur de s++ : %d et de us++ : %d\n", s, us);
  printf("Représentation de s : %x et de us : %x\n", s, us);
}
```

```
2) Gestion des aspects sign◆s ou non-sign◆s :
Un nombre : 0xffffffff mais deux Interpretations
        => sign◆e -1 et non sign◆ 4294967295

Valeur de s : 32767 et de us : 32767
Meme repr◆sentation binaire de s : 7fff et de us : 7fff
On incr◆mente l'◆volution diff◆re (car les types diff◆rent) !
Valeur de s++ : -32768 et de us++ : 32768
Repr◆sentation de s : ffff8000 et de us : 8000
Valeur de s++ : -32767 et de us++ : 32769
Repr◆sentation de s : ffff8001 et de us : 8001
```

Les entiers : Largeur machine

Au moment du choix entre les différents types entiers, il faut savoir que dans un processeur **les calculs sont faits dans des registres**.

➢ Or, les registres ont une taille fixe indépendante de la taille des données qu'ils manipulent.

Règle de promotion :

La conséquence est qu'il **n'y a pas** de calcul sur des valeurs plus petites qu'un registre, donc qu'un int (puisque ce dernier est censé représenter la précision naturelle de la machine).

Les réels

L'écriture d'un nombre réel laisse présager la complexité de sa représentation :

$$\text{Nombre d'Avogadro} : 6,02214129x10^{23}mol^{-1}$$

Certains réels ont plusieurs décimales, d'autres pas :

- ➢ Il y a au moins une virgule dont la position peut varier.
- ➢ Il peut avoir un signe,
- ➢ On peut utiliser un exposant.

La difficulté n'est pas la représentation en base 2 qui admet aussi une virgule :

$$\begin{aligned} -0,40625 &= 0*2^0+0*2^{-1}+1*2^{-2}+1*2^{-3}+0*2^{-4}+1*2^{-5} \\ &= -0,01101*2^0 \\ &= -1,101*2^{-2} \end{aligned}$$

La difficulté est dans **le peu de bits disponibles (32, 64, 128) pour représenter un support continu infini** puis **dans le codage.**

Les flottants

Les flottants se veulent être l'abstraction informatique des réels en mathématiques :

- Les réels ($\mathbb{R}$) sont à support continu.
- La notion de FLOTTANT est déduite d'un encodage sur 32, 64, 128 bits, expliquant qu'il ne peut y avoir continuité :

$$(1.0/3.0 + 1.0/3.0 + 1.0/3.0) \neq 1$$

Il y a en C (K&R) deux types de flottants, pour deux précisions :

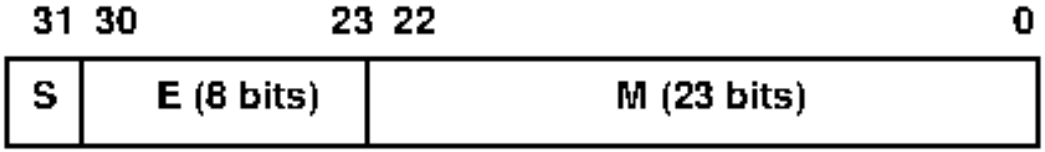
- La simple sur 4 octets : **float**
- La double sur 8 octets : **double**

La norme C ANSI en a ajouté une, dans le but d'être capable d'exploiter les évolutions matérielles permettant des précisions étendues au niveau des microprocesseurs :

long double

Les flottants : ANSI/IEEE 754 (1985)

Les flottants sont généralement implantés selon cette norme :



La valeur d'un float est donné par la formule (en simple précision) :

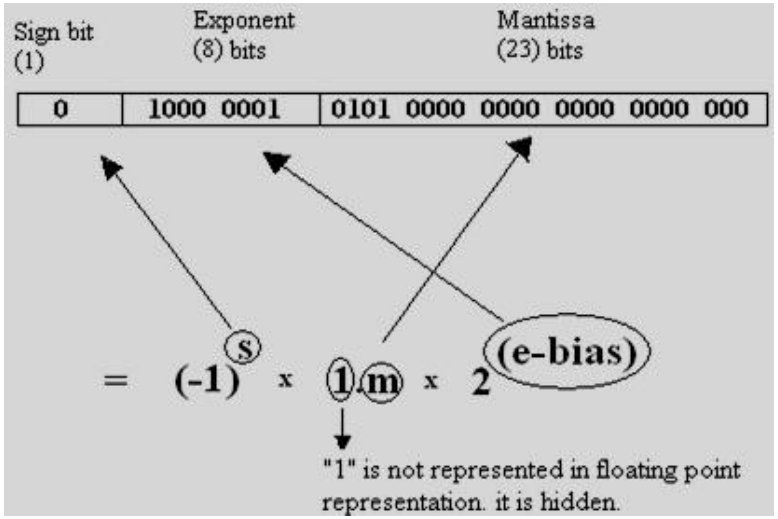
- $-1^{Signe} * 1.Mantisse * 2^{Exposant-127}$

Par exemple :

$$\begin{aligned} -0,40625 &= 0 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} + 1 * 2^{-3} + 0 * 2^{-4} + 1 * 2^{-5} \\ &= -0,01101 * 2^0 \\ &= -1,101 * 2^{-2} \end{aligned}$$

- Signe = 1 car valeur négative
- Mantisse = 10100000 car le « 1, » est implicite en IEEE
- Exposant : -2 + 127 (biais soustrait lors du calcul de la valeur) soit 125 (01111101_2 / $7D_{16}$)

Exemple de codage IEEE754



S(signe)= 0, E(exposant biaisé) = 129, M= 1,0101 . . .

bias $= 2^{(8-1)}-1$

$= 127$

Decimal value $= (-1)^0 \times 1.0101\ 0000\ 0000\ 0000\ 0000\ 000 \times 2^{(129-127)}$

$= 1.0101\ 0000\ 0000\ 0000\ 0000\ 000 \times 2^{(2)}$

$= 101.01\ 0000\ 0000\ 0000\ 0000\ 000$ (Binary fraction)

↓ Convert binary fraction to decimal

$= (1x2^2)+(0x2^1)+(1x2^0) . +(0x2^{-1}) + (1x2^{-2}) + (0x2^{-3}) \ldots\ldots (0x2^{-21})$

$= (4+0+1+0+0.25)$

$= 5.25$

Printf . . . ieee754

```c
/* printwith_a_specifier : print to show ieee 754 */
#include <stdio.h>
#include <float.h> /* pour DIG */
#include <inttypes.h> /* pour PRIx64 */
int main(void){
  float fv;
  double dv;
  /* DIG : the number of significant digits a string may be scanned
     into a floating point and then the FP printed, still retaining
     the same string presentation */

  fv = -0.40625; /* En float ------------------------------------- */
  /* Variable-width conversion specifier (".*") */
  printf("fv = %.*e \n" , FLT_DIG, fv); /* given  FLT_DIG */
  printf("\t hexa  \t0x%x\n", *((int  *)(&fv)));
  /* use of "a" specifier */
  printf("\t ieee  \t%a\n",fv);

  fv = 5.25;
  printf("\nfv = %.*e \n" , FLT_DIG, fv);
  printf("\t hexa  \t0x%x\n", *((int  *)(&fv)));
  printf("\t ieee  \t%a\n",fv);

  dv = -0.40625;  /* En double ------------------------------------ */
  printf("\ndv = %.*e \n" , DBL_DIG, dv);
  printf("\t hexa  \t0x%"PRIx64" \n", *((long int *)(&dv)));
  printf("\t ieee  \t%a\n",dv);

  dv = 5.25;
  printf("\ndv = %.*e \n" , DBL_DIG, dv);
  printf("\t hexa  \t0x%"PRIx64" \n", *((long int *)(&dv)));
  printf("\t ieee  \t%a\n",dv);
  return 0;
}
```

```
$ ./a.out
fv = -4.062500e-01
        hexa view       0xbed00000
        ieee view       -0x1.ap-2

fv = 5.250000e+00
        hexa view       0x40a80000
        ieee view       0x1.5p+2

dv = -4.062500000000000e-01
        hexa view       0xbfda000000000000
        ieee view       -0x1.ap-2

dv = 5.250000000000000e+00
        hexa view       0x4015000000000000
        ieee view       0x1.5p+2
```

Utilisation des flottants

Intrinsèquement à la norme, plusieurs problèmes d'utilisation se posent :

- **Le support des valeurs représentables n'est pas continu puisque numérique** et donc chaque valeur est un code de **taille fixe**.

  Par exemple, le nombre rationnel $(1/7)_{10}$ sera approximé :

```
1  #include <stdio.h>
2  #include <inttypes.h> /* pour PRIx64 */
3  #include <float.h>
4
5  int main(void){
6    double x = 0.9375;
7    printf("\t hexa  \t0x%"PRIx64" \n", *((long int *)(&x)));
8    printf("\t ieee  \t%a\n",x);
9
10   int Digs = DBL_DIG;
11   double OneSeventh = 1.0/7.0;
12   printf("%.*e\n", Digs, OneSeventh); // 1.428571428571428492127e-01
13
14   return 0;
15 }
```

  Bien sûr, le fait que son développement théorique sous forme décimale soit illimité pose un problème pour une machine.

➢ L'arithmétique binaire pour représenter des nombres en base 10 accentue encore ce manque d'infini . . .

Le nombre 0.1_{10} nécessiterait un nombre infini de symboles binaires pour être représenté (sauf arrondi) :

$$0.1_{10} = 0.0001100110011001100110011001100110011001100111\ldots_2$$

La machine ne dispose que de 32, 64 ou 128 bits en lieu et place de l'infini !

Range, Overflow et Not a Number

Le **support est borné** puisque le codage utilise un nombre fixe de bits.

```c
/* floatover.c */
#include <stdio.h>
#include <float.h>
int main(void){
  float f;
  printf("Storage size for float : %zu \n", sizeof(f));
  printf("Precision value: %d\n", FLT_DIG );

  f = FLT_MIN; /* Smallest number without losing precision */
  printf("f = %.*e\t(0x%x)\n", FLT_DIG, f,*((int *)(&f)));

  f = f/10.0;  /* Plus petit MAIS avec perte de precision */
  /* cf http://www.cprogramming.com/tutorial/floating_point/
understanding_floating_point_representation.html*/
  printf("f = %.*e\t(0x%x)\n", FLT_DIG, f,*((int *)(&f)));

  f = FLT_MAX; /* Largest representable number*/
  printf("f = %.*e\t(0x%x)\n", FLT_DIG, f,*((int *)(&f)));

  f = f*10.0;   /* On provoque un overflow */
  printf("f = %.*e\t(0x%x)\n",FLT_DIG, f,*((int *)(&f)));

  /* On cree l'infini */
  f = f/0.0;
  printf("f = %.*e\t(0x%x)\n",FLT_DIG, f,*((int *)(&f)));

  /* On provoque un NaN */
  f = 0.0/0.0;
  printf("f = %.*e\t(0x%x)\n",FLT_DIG, f,*((int *)(&f)));
  return 0;
}
```

```
menez@vtr /Users/menez/EnseignementsC
$ gcc floatover.c

menez@vtr /Users/menez/EnseignementsC
$ ./a.out
Storage size for float : 4
Precision value: 6
f = 1.175494e-38        (0x800000)
f = 1.175495e-39        (0xccccd)
f = 3.402823e+38        (0x7f7fffff)
f = inf (0x7f800000)
f = inf (0x7f800000)
f = -nan        (0xffc00000)
```

Précision des flottants

```c
/* scalf.c : Variables flottantes */
#include <stdio.h>

int main(int argc, char *argv[]){
  float a;   /* simple précision */
  double b;  /*  double précision */

  b = 0.1;
  a = b;
  printf("\nPour une variable de type float sur %zu octets", sizeof(float));
  printf("\nLa précision entre 6 et 9 décimales :  %20.15f \n",a);
  /* https://en.wikipedia.org/wiki/Single-precision_floating-point_format*/

  printf("\nPour une variable de type double sur %zu octets", sizeof(double));
  printf("\nLa précision est de l'ordre de 15 décimales :  %20.15f \n",b);
  /* https://en.wikipedia.org/wiki/IEEE_floating_point */
  printf("\nAu delà ??? :  %20.20f \n",b);
}
```

```
menez@vtr /Users/menez/EnseignementsCurrent/CodeC
$ ./a.out

Pour une variable de type float sur 4 octets
La précision entre 6 et 9 décimales :     0.100000001490116

Pour une variable de type double sur 8 octets
La précision est de l'ordre de 15 décimales :     0.100000000000000

Au delà ??? :  0.10000000000000000555
```

Voir les sites Web mentionnés pour une définition de la précision . . .

Précision des flottants

La précision dépend de la valeur du nombre représenté :

➢ La **densité des valeurs représentables est plus forte vers 0**.

Ceci afin d'avoir des erreurs d'arrondis relativement (à la valeur du nombre) constantes.

```
1   #include <stdio.h>
2   #include <inttypes.h> /* pour PRIx64 */
3   #include <float.h>
4   int main(void){
5     float r = 0;
6
7     /* For example, float can easily distinguish between 0.0 and 0.1.*/
8     printf( "%.6f\n", r ) ; // 0.000000
9     r+=0.1 ;
10    printf( "%.6f\n", r ) ; // 0.100000
11
12    /*But float has no idea of the difference between 1e27 and 1e27 + 0.1.*/
13    r = 1e27;
14    printf( "%.6f\n", r ) ; // 999999988484154753734934528.000000
15    r+=0.1 ;
16    printf( "%.6f\n", r ) ; // still 999999988484154753734934528.000000
17
18    return 0;
19  }
```

Tous les bits de la mantisse sont alors utilisés pour représenter la partie entière (gauche) du nombre !

Tester l'égalité . . . Danger !

```
1   /* floatprec.c */
2   #include <stdio.h>
3
4   #define MAX 0.5
5   int main(void){
6     float t = 0.0;
7
8     printf("Début !\n");
9     while ( t != MAX){
10      t = t + 0.1;
11      printf("t = %f\n",t);
12    }
13    printf("Fin !\n");
14
15    return 0;
16  }
```

```
menez@vtr /Users/menez/Ense
$ ./a.out
D◆but !
t = 0.100000
t = 0.200000
t = 0.300000
t = 0.400000
t = 0.500000
Fin !
```

QUIZ : Que se passe t'il si on modifie la valeur de MAX . . . 1.0 ?

Why ?

```
1   /* floatcompteurinit.c */
2   #include <stdio.h>
3
4   int main(void){
5     float t = 16600000.;
6     float inc = 1.0;
7     printf("Début !\n");
8     while (t<16800000.0){
9       t = t + inc;
10      printf("t = %f\t(0x%x)\n",t,*((int *)(&t)));
11    }
12    printf("inc = %f\t(0x%x)\n",inc,*((int *)(&inc)));
13    printf("Fin !\n");
14
15    return 0;
16  }
```

Essayer donc ce code "tout simple" . . .

Pour vous aider . . .

```
1   /* floatcompteur.c */
2   #include <stdio.h>
3
4   int main(void){
5     float t = 16600000.;
6     float inc = 1.0;
7     printf("Début !\n");
8     while (t<16777216.0){
9       t = t + inc;
10      printf("t = %f\t(0x%x)\n",t,*((int *)(&t)));
11    }
12    t = t + inc;
13    printf("t = %f\t(0x%x)\n",t,*((int *)(&t)));
14    t = t + inc;
15    printf("t = %f\t(0x%x)\n",t,*((int *)(&t)));
16    t = t + inc;
17    printf("t = %f\t(0x%x)\n",t,*((int *)(&t)));
18
19    printf("inc = %f\t(0x%x)\n",inc,*((int *)(&inc)));
20    printf("Fin !\n");
21
22    return 0;
23  }
```

```
/Users/menez/EnseignementsCurrent/Co
t = 16777201.000000     (0x4b7ffff1)
t = 16777202.000000     (0x4b7ffff2)
t = 16777203.000000     (0x4b7ffff3)
t = 16777204.000000     (0x4b7ffff4)
t = 16777205.000000     (0x4b7ffff5)
t = 16777206.000000     (0x4b7ffff6)
t = 16777207.000000     (0x4b7ffff7)
t = 16777208.000000     (0x4b7ffff8)
t = 16777209.000000     (0x4b7ffff9)
t = 16777210.000000     (0x4b7ffffa)
t = 16777211.000000     (0x4b7ffffb)
t = 16777212.000000     (0x4b7ffffc)
t = 16777213.000000     (0x4b7ffffd)
t = 16777214.000000     (0x4b7ffffe)
t = 16777215.000000     (0x4b7fffff)
t = 16777216.000000     (0x4b800000)
t = 16777216.000000     (0x4b800000)
t = 16777216.000000     (0x4b800000)
t = 16777216.000000     (0x4b800000)
inc = 1.000000  (0x3f800000)
Fin !
```

QUIZ : Pourquoi l'incrément n'a t-il plus lieu ?

Les flottants et les implantations matérielles

Tableau des tailles : sizeof()

Linux@iPentium	Solaris Ultra-Enterprise
int=4, long=4, long long=8, float=4, double=8 long double=12	int=4, long=4, long long=8 float=4, double=8 long double=16

« The Pentium floating-point unit (FPU) performs operations on 32-, 64-, and 80-bit operands. »

- Pentium floating-point operations conform to IEEE standards 754 and 854.
- When a floating-point operand is loaded into the FPU, it is converted to 80 bits and all operations inside the FPU are carried out using an 80-bit data width.

Les flottants et le compilateur

Les mêmes précautions de portabilité que pour les **int** s'appliquent !

- Extrait de Microsoft (MSDN) :
  « The long double data type (80-bit, 10-byte precision) is mapped directly to double (64-bit, 8- byte precision) in Windows NT, Windows 98, and Windows 95. »
- Ce qui n'est pas le cas dans Linux qui réalise les arrondis en précision étendue.

Beaucoup d'informations sont disponibles dans <limits.h> et <float.h>

- Concernant les valeurs minimales et maximales représentables,
- Concernant la configuration du contrôleur IEEE754 (mode d'arrondi, . . .).

Les énumérations : **enum**

Enumération :

Une énumération est un type entier (unsigned int) dans lequel certaines valeurs sont associées à des constantes symboliques représentées par des identificateurs.

Par exemple :

```
enum couleur {rouge, vert, bleu, blanc, jaune} c;
enum quadrilatere {carre, rectangle, losange,
                   parallelogramme} q1, q2;
```

Un tel exemple définit :

① un type énuméré nommé `enum couleur`

② et une variable (c) correspondant à cette énumération.

③ un type énuméré nommé `enum quadrilatere`

④ et deux variables (q1 et q2) correspondant à cette énumération.

Enumération : utilisation

Dans la suite du programme, les expressions `enum quadrilatere` et `enum couleur` se comportent comme des identificateurs de type et peuvent être utilisées comme tel.

```
enum couleur fond;
enum quadrilatere x,y;
```

Les valeurs définies entre accolades constituent le domaine de valeurs :

```
fond = blanc;
x = carre;
```

Enumération : attention !

L'implémentation en C des énumérations, attribue par défaut les valeurs entières : $[0, 1, 2, \ldots, n]$ aux différentes valeurs énumérées du domaine.

➢ Ainsi, « rouge » vaut 0, « vert » vaut 1, « losange » vaut 2 . . .

On peut changer la valeur de ces constantes symboliques :

```
enum escape {TAB='\t', NEWLINE='\n', RETURN='\r'};
enum mois {jan=1, fev, mars, avril, mai, juin, juillet};
```

Les noms des constantes figurant dans les énumérations (d'une même portée) doivent être différents. Voici des exemples non valides :

```
enum etudiants_L2 { martin, dupond, robert, martin};
enum admis { martin, durand};
```

Enumération : pourquoi ?

Dans le cas de l'abstraction des « booléens », on énumère les différentes valeurs possible de b : true, false.

```
enum boolean {false, true} b;
```

Une solution basée sur le préprocesseur ne permettrait pas de conserver le typage :

```
#define true 1
#define false 0
int b = false;
b = 9;  /* Que penser de cette affectation ? */
```

Malheureusement, « gcc laisse passer :-(((»

void

Ce type permet de désigner des objets qui n'ont pas de taille.

Il sert :

① comme type de résultat des fonctions qui ne renvoient pas de valeur (l'équivalent en C des *procédures* du langage Pascal ou *subroutines* en FORTRAN),

```
void tri_par_insertionf (int T[], int N){
```

② pour prototyper les fonctions sans paramètres,

```
int f(void)
```

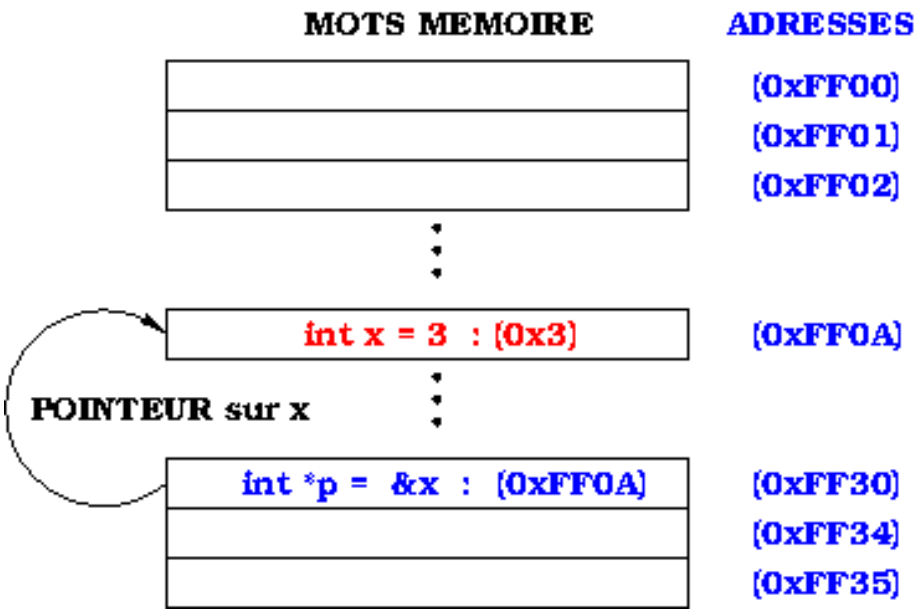
③ pour définir un type générique pour les pointeurs.

```
void *p; /* Débrayage de type */
void x;  /* Ceci est NON VALIDE !*/
```

Les pointeurs : *

La notion de pointeur permet de manipuler des valeurs d'adresses mémoire.

➢ Ces adresses étant celles de variables préalablement définies.
Le pointeur est **une référence à** une variable.



p est une variable pointeur qui contient l'adresse de la variable **x**, eg. une référence à **x**.

➢ On peut alors **manipuler x au travers de p** !

Les pointeurs : le type pointé

La notion de pointeur **nécessite d'être formé avec le type de l'objet pointé**.

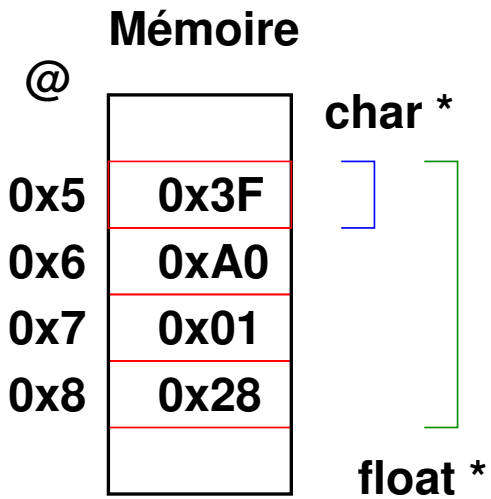
Ce peut être :

- un type de base :
  char *p ou **float *p** ou . . .
- un type agrégat ou construit :
  struct image * p ou **image *p**
- un pointeur (ça devient un pointeur de pointeur) :
  int **p
- une fonction :
  char (*f)(int x)
- **void** pour désigner une adresse générique, qui est la façon élégante de dire que l'on débraye momentanément le contrôle de type :
  void *p

Pourquoi "le type pointé" ?

L'adresse d'un objet ne suffit pas au compilateur pour accéder correctement à l'objet.

➢ Il a besoin aussi de sa structure : son type !



0x5 est l'adresse :

➢ de la valeur 0x3F si son type est char *

➢ de la valeur 0x3FA00128 si son type est float *

Créer un nom de type : **typedef**

Le langage C fournit un opérateur (mot clé : **typedef**) servant à créer des noms de nouveaux types de données.

Par exemple, la déclaration :

```
typedef int Longueur ;
```

rend le nom `Longueur` synonyme de `int`.

On peut ensuite employer le type `Longueur` exactement de la même façon que le type `int` :

```
int x;
Longueur y;
```

Une déclaration typedef **ne crée en aucune façon un nouveau type**.

- ➢ Elle "ajoute" simplement un nouveau nom désignant un type (de base ou construit) existant.

Définir ainsi des synonymes permet de considérablement simplifier les notations et augmente la lisibilité des programmes.

Exemples d'utilisations de typedef

```
/* un type : ushort */
typedef unsigned short ushort;

/* un type : matrix */
typedef int matrix[100][100];

/* deux types : complexe et p_complexe */
typedef struct comp {
  float reel;
  float imag;}  complexe, *p_complexe;

/* un type boolean : enum */
typedef enum {false,true} boolean;

/* un type fdiadique : pointeur sur fonction */
typedef void (*fdiadique) (int a, int b);
```

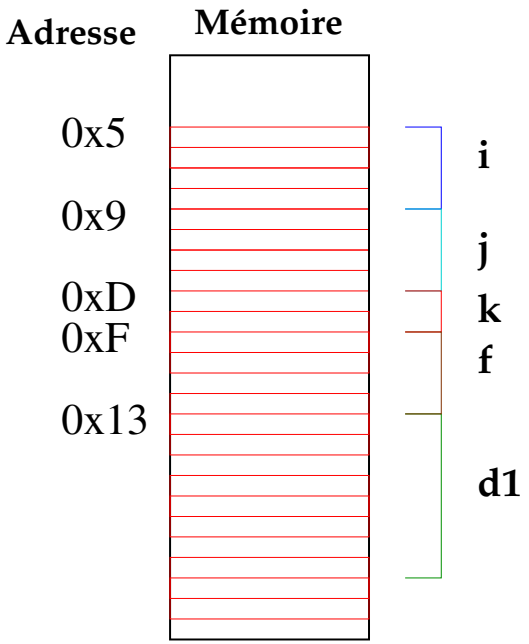
Sémantique d'une définition de variable

La **définition de variable** vise à associer un nouvel objet en mémoire à un identificateur :

- ➢ Il y a donc **allocation** d'un espace mémoire.
- ➢ Cet espace mémoire porte désormais un nom qui n'identifie que lui !

Exemples :

```
int i,j;
short int k;
float f;
double d1;
```



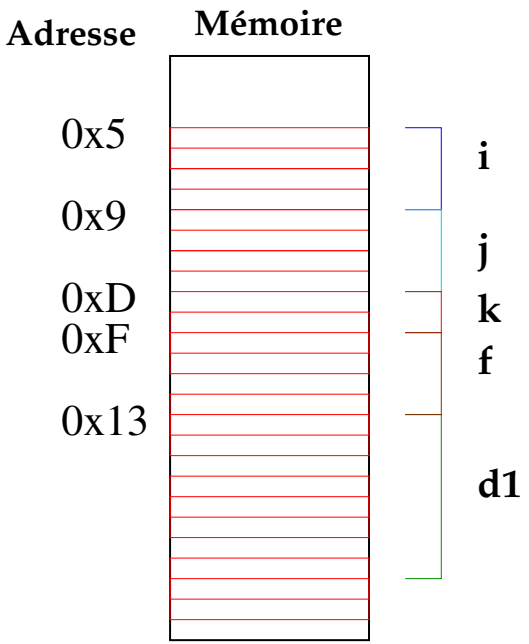
Syntaxe d'une définition de variable

Une telle définition se fait **en faisant suivre le nom du type, par la liste des noms des variables :**

[classe_allocation] nom_type identificateur ... ;

Exemples :

```
int i,j;
short int k;
float f;
double d1;
```



Le choix du type dit ce qu'est la variable . . . en termes de taille, de codage, . . .

➢ C'est le **rôle du programmeur** de définir en mémoire autant de variables qu'il estimera nécessaire pour l'accomplissement correct de son programme :

- ✔ mémoriser des résultats intermédiaires,
- ✔ des résultats définitifs,
- ✔ des indices de tableaux,
- ✔ . . .

➢ C'est le **rôle du compilateur** de mettre en place un mécanisme de gestion mémoire capable d'attribuer automatiquement à l'exécution, une adresse à une telle variable.

- ✔ pile d'exécution,
- ✔ tas/heap et fonctions d'allocation,
- ✔ . . .

Identificateur

Tout identificateur (variables, fonctions, . . .) doit répondre à plusieurs critères syntaxiques :

On ne peut pas choisir n'importe quoi !

- Un identificateur est une séquence de lettres et de chiffres.
- Le premier caractère doit être une lettre.
- Le caractère de soulignement « _ » compte comme une lettre.
- Les lettres majuscules et minuscules sont différenciées.
- Ils peuvent être de longueur quelconque mais seuls les 31 premiers caractères sont significatifs.

Il peut aussi répondre à des critères de « bon sens » !

- Eviter les « a », « b », « c » sauf s'il s'agit des coefficients d'une équation du second degré !
- Il est important que l'identificateur **aide à la compréhension du code**.

Le qualificatif : **restrict**

```
void f1(int n,
        float * restrict a1,
        const float * restrict a2)
{
  int i;
  for ( i = 0; i < n; i++ )
     a1[i] += a2[i]; /* a1 et a2 ne référencent pas
                        les mêmes zones mémoires */
}
```

Alignement

Pointeurs : Initialisation

Une vision simple d'un objet . . .

On nomme usuellement « objet » toute donnée constituée d'un certain nombre d'octets consécutifs en mémoire : une zone en mémoire.

Le langage C permet d'accéder et de manipuler **les adresses des objets** (eg. des références aux objets) au travers de deux opérateurs :

① l'opérateur unaire `&` qui permet d'obtenir une **référence** sur un objet.

```
& objet
```

Pour parler de cette action de référenciation, on peut trouver les termes :

« obtenir **un pointeur sur l'objet** »/« obtenir l'adresse de l'objet »

② l'opérateur unaire `*` qui permet de **déréférencer** une référence pour obtenir l'objet « au bout » de la référence.

```
*adresse_objet
```

Avec la même sémantique, on peut dire que **la déréférenciation représente la valeur contenue** dans l'objet pointé ou encore :

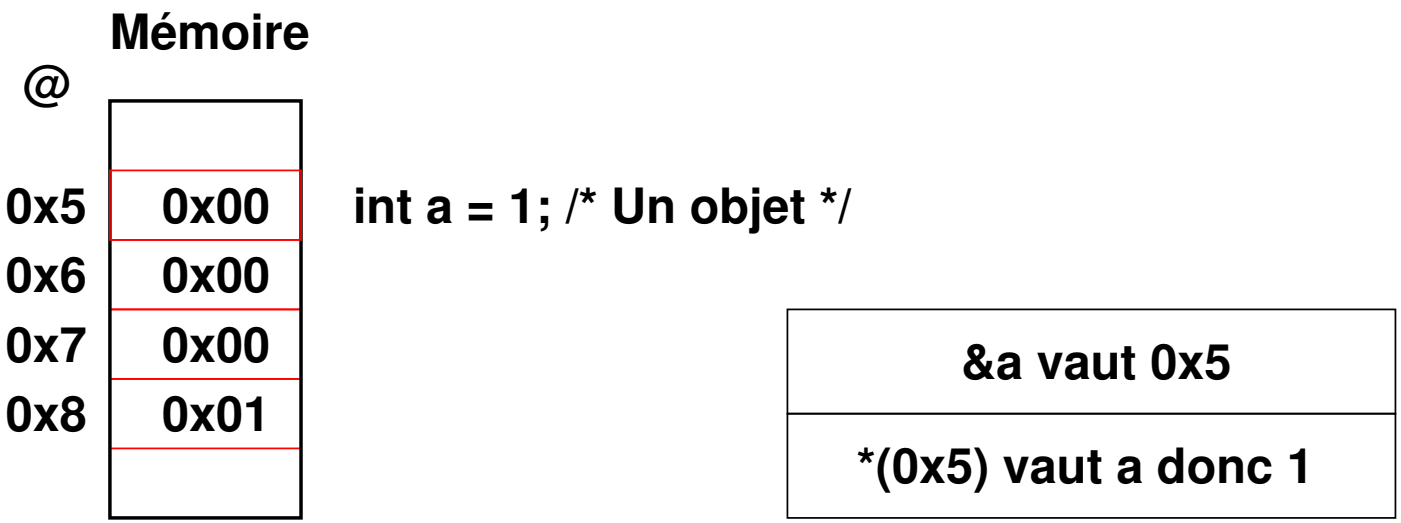
"le pointeur permet ainsi d'accéder à la valeur de l'objet pointé".

L' **intérêt de la notion de pointeur** vient du couplage à la notion de variable de pointeurs.

- Selon la référence qu'elle contient, cette variable pointeur permet alors d'accéder à différentes variables . . . un même identificateur permet de manipuler différentes zones mémoires. Ceci est impossible sans les pointeurs
- On y reviendra notamment dans le passage de références lors des appels de fonctions.

Constante de pointeur

L'expression obtenue par l'opérateur `&` est une **constante de pointeur** (en ce sens qu'elle ne peut être modifiée).



Typage

On a déjà vu dans le cadre des types de base, que le type adresse devait nécessairement (sauf exception void *) exprimer le type de la zone mémoire pointée.

```
int *          /* type pointeur vers un int          */
double *       /* type pointeur vers un double       */
void *  /* type pointeur vers n'importe quel type */
```

En toute logique, les deux opérateurs & et * créent de nouvelles valeurs de types bien définis :

Si le type d'une référence p est t *, alors le type de *p est t.

➢ Bien sûr, *(&x) est équivalent à x c'est-à-dire que la prise de référence et la déréférenciation sont inverses (dans cet ordre !).

Les variables de pointeurs

On peut définir des **variables (zones mémoires) dont la valeur sera une adresse**.

La définition d'une **variable** de nom X correspondant à un pointeur sur un objet de type **type** sera réalisée par :

type *X

Une telle définition exprime le fait que **le type de l'objet *X** est ***type*** c'est-à-dire que :

- ➢ *X* est un objet dont la valeur, une adresse, pointe sur un élément de type *type*.

En aucun cas, la définition d'une variable pointeur **NE RESÉRVE** d'espace mémoire pour l'objet pointé.

Exemple

```
1  int a = 3;
2  int *pa;    /* Pointeur sur int */
3  int **ppa; /* Pointeur de pointeur sur int */
4  float f;
5  float *pf;
6  /* Référenciation et affectation */
7  pa = &a;
8  printf("a vaut %d\n",a);
9  /* Déréférenciation en rvalue*/
10 printf("a vaut %d\n",*pa);
11 ppa = &pa;
12 /* Déréférenciation de dereferenciation */
13 printf("a vaut %d\n",*(*ppa));
14 pf = &f ;
15 /* Déréférenciation en lvalue */
16 *pf = 3.0/4.5;
```

La valeur NULL

NULL

La constante `NULL` est macro-définie dans le fichier standard <stdio.h>.

Elle correspond à une valeur « non définie » pour les pointeurs :

- ➢ car illégale dans l'espace d'adressage.

Ainsi, de nombreuses fonctions standard renvoient NULL, si leur déroulement a été perturbé.

ça ne se discute pas !

- ➢ Toute variable de pointeur dont le contenu, une adresse, n'est pas « valide », doit être initialisée avec la valeur NULL.

Instruction **goto**

Toute instruction peut être précédée d'un identificateur suivi du signe « : »

- Cet identificateur est appelé **étiquette** .
- Une instruction « goto identificateur ; » a pour effet de transférer le contrôle d'exécution à l'instruction étiquetée par identificateur.

```
int i,j, erreur;
for (i=0 ; i<N ; i++){
   for (j=0 ; j<N ; j++){
      if (erreur == VRAI) goto traiterr;
         ...
   }
traiterr :  /* Etiquette dans le corps de la fonction */
   printf(''Erreur !'');  ...
```

Attention : l'étiquette doit être dans la fonction en cours !

- Elle peut être positionnée avant ou après l'utilisation.

Instruction ;

Cette (**instruction nulle**) ne rend que des services syntaxiques.

① Elle peut être utile quand on désire mettre une étiquette à la fin d'une instruction composée.

```
    goto fin;
    ...
    fin: ;
}
```

② Elle est également utile pour mettre un corps nul à certaines instructions itératives.

```
for (i = 0; i < N; t[i++]= 0)
      ; /* instruction nulle */
```

Remarque :

On a vu les effets désastreux dans le cas où elle est mal placée dans les instructions d'itération.

Fonctions

Le concept de fonction :

La fonction permet de nommer un ensemble d'instructions.

➢ C'est donc l'unité de structuration syntaxique au-dessus des instructions.

```
double delta(double a, double b, double c){
  /* rend la valeur du discriminant */
  return b*b - 4*a*c;
}
...
d = delta(1.0, 3.0, 1.0);
...
```

Attention : En C, on ne peut pas définir des fonctions dans des fonctions. Elles sont toutes "au niveau 0".

Intérêts du concept de fonction

```
if (b*b-4*a*c > 0){
   ...
}
if (b*b-4*a*c == 0){
   ...
}
if (b*b-4*a*c < 0){
   ...
}
```

① La décomposition du traitement qu'induit la fonction permet de **factoriser des traitements** et ainsi de **lutter contre la duplication** (et donc la redondance) de codes.

Cela induit que le code ainsi factorisé pourra s'exécuter avec des valeurs de variables différentes : **les paramètres**.

② C'est aussi un outil permettant de **composer une abstraction** (voir le chapitre sur la modularité et votre expérience de la programmation objet).

Fonctions

Les fonctions **se déclarent, se définissent et s'appellent**.

- Il doit bien sûr y avoir compatibilité entre leur définition et leurs appels (nombre et type des arguments, type de retour)
- Le rôle de la déclaration est de permettre au compilateur de vérifier cela.

Les fonctions sont enfin **un lieu de localisation de variables** :

- Certaines de ces variables verront leur initialisation réalisée lors de l'appel : **paramètres formels**.
- D'autres sont gérées (initialisation et évolution) complètement localement par la fonction et le calcul qu'elle développe : **variables locales** (non initialisées par le mécanisme d'appel de la fonction).

Définition d'une fonction

Syntaxe :

```
type identificateur ( [déclarations-de-variables1] ){
   [déclarations-de-variables2]
   instructions
 }
```

Exemple :

```
/* Fonction en ANSI : ----------*/
int maxelement(int T[], int N) {
  /* rend l'index de l'element max du tableau T */
  int i,max = 0;
  for (i=0; i<N ; i++)
    if (T[i]>T[max])
      max = i;
  return max;
}
```

Sémantique de la définition d'une fonction

① **type** est le type de la valeur rendue par la fonction.

② **identificateur** est le nom de la fonction.

③ **déclarations-de-variables1** est la liste des déclarations des paramètres formels permettant d'indiquer leur position, leur nom et leur type.

④ **déclarations-de-variables2** permet, si besoin est, de déclarer des variables qui seront "locales" à la fonction, elles seront donc inaccessibles de l'extérieur (c'est-à-dire des autres fonctions).

⑤ Parmi les **instructions** exécutées sur appel de la fonction, il peut y avoir au moins une instruction du type :

return expression ;

L'expression est évaluée, et le contrôle d'exécution est rendu à l'appelant.

➢ La "valeur rendue par la fonction" est celle de expression.

S'il n'y a pas de « return », alors le type de la fonction doit être `void` .

Syntaxe K&R : les reliques ...

```
/* Fonction en ANSI : ----------*/
int maxelement(int T[], int N) {
  int i,max = 0;
  for (i=0; i<N ; i++)
    if (T[i]>T[max])
      max = i;
  return max;
}

/* La même fonction en K&R : --*/
maxelement(T,N)
int T[];
int N;
{  /* Renvoie int par défaut --*/
  int i,max = 0;
  ....
```

Appel de fonction

L'"appel de fonction" permet d'exécuter le code factorisé dans la fonction.

Syntaxiquement, un appel de fonction est réalisé lorsqu'un nom de fonction est suivi d'une parenthèse ouvrante :

- cet appel peut être l'opérande d'une expression :

$$i = j + pow(i, j)\,;$$

- ou une instruction à part entière :

printf("toto") ;

Remarque :

A l'exécution, il n'y a **aucun contrôle dynamique** ni sur le nombre, ni sur le type des paramètres ...

- d'où l'intérêt du prototypage qui permet un contrôle statique (à la compilation).

Un tel appel entraîne :

```
int f (int x){
  int y = x+1;
  return y;
}
int main(void){
  int a = 1;
  a = f(3*4);
}
```

① l'évaluation des paramètres d'appel :

➢ l'ordre d'évaluation est **non normalisé** !

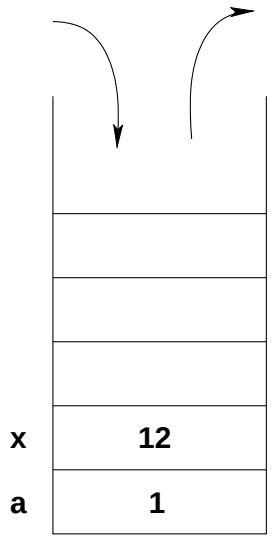
② pour chaque paramètre, il y a création sur la pile d'exécution d'une variable **locale à l'appel** qui sera supprimée au retour de l'appel.

Cette variable est initialisée à la valeur de l'expression correspondante (par sa position) qui constitue le paramètre effectif :

➢ c'est la **TRANSMISSION PAR VALEUR** des paramètres.

Pile d'exécution

La mémoire est une « étagère », un tableau.



Rien n'empêche de gérer ce tableau comme une pile :

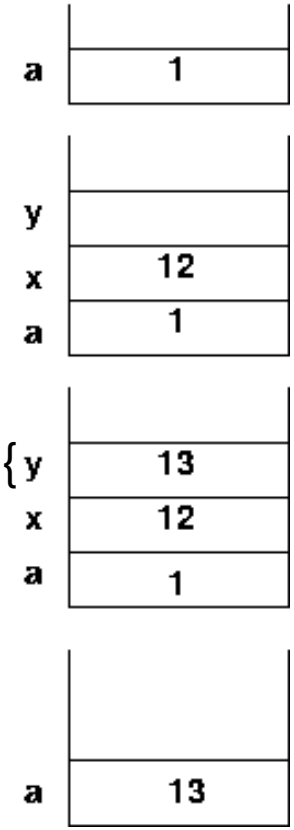
- On crée et on supprime dans la mémoire des éléments, par le sommet de la pile.

C autorise la récursivité :

Un langage, qui ne mettrait pas en place au niveau de la gestion mémoire ce modèle d'exécution, ne pourrait pas gérer de récursivité !

Passage de paramètres par valeur

```
int f (int x){
  int y = x+1;
  return y;
}
int main(void){
  int a = 1;
  a = f(3*4);
}
```



A l'appel de `f` par `main`,

① $3 * 4$ est évalué, donnant 12,

② x et y sont créés sur la pile,

③ et x est initialisé avec 12,

④ les instructions de la fonction sont exécutées : y vaut 13,

⑤ la valeur de y est celle renvoyée par la fonction

⑥ et a est affectée avec cette valeur.

⑦ La fonction prenant fin, x et y son retirées de la pile.

L'existence (i.e. la présence) de x et de y sur la pile est liée au fait que la fonction f est en cours d'exécution !

➢ Variables automatiques

Valeur de retour

L'instruction **return** est une instruction comme une autre, il est donc possible d'en utiliser autant qu'on le désire dans le corps d'une fonction.

```
int max(int i,int j){ /* Calcul de la valeur max */
  if (i > j)
    return i;
  else
    return j;
}
```

Dans le cas où la dernière instruction exécutée par une fonction n'est pas une instruction return, la valeur renvoyée par la fonction est **indéterminée** :

- Soit la fonction ne devait rien renvoyer (void).
- Soit il s'agit d'une erreur de programmation.

Ne surtout pas utiliser une notation fonctionnelle pour appeler la fonction dans ce cas !

Procédures

Le langage C **ne comporte pas** de concept de procédure.

```
tri_par_selection(X,nb);
```

Cependant, les fonctions pouvant réaliser sans aucune restriction tous les effets de bord qu'elles désirent,

- ➢ le programmeur peut **réaliser une procédure à l'aide d'une fonction qui ne renverra aucune valeur**.

```
void tri_par_selection(int T[],int N) {
  ...
  return; /* Gestion du flot */
}
```

Procédures (suite)

```
void tri_par_selection(int T[],int N) {
  ...
  return; /* Gestion du flot */
}
```

Une procédure sera donc implémentée :

① sous la forme d'une fonction renvoyant *void*

② et dont la partie liste-d'instructions ne comportera pas d'instruction « return expression ; »

➢ mais éventuellement des instructions « return ; »

Lors de l'appel de la procédure, il faudra ignorer la valeur renvoyée

➢ c'est-à-dire ne pas l'englober dans une expression.

```
tri_par_selection(X,nb);
```