IGI 2101

Introduction à la programmation en langage C

Thierry Grandpierre

Bureau 5256

ESIEE

(thierry.grandpierre@esiee.fr)

Plan du cours

1. Introduction
2. Premier contact avec le C
3. Les types et structures de données
4. Fonctions
5. Pointeurs
6. Bibliothèque d’entrées sorties
7. Allocation mémoire dynamique
8. Chaîne de compilation, compil. séparée, edit. de liens dyna.
9. Pour aller plus loin
10. Bibliographie

I-Introduction

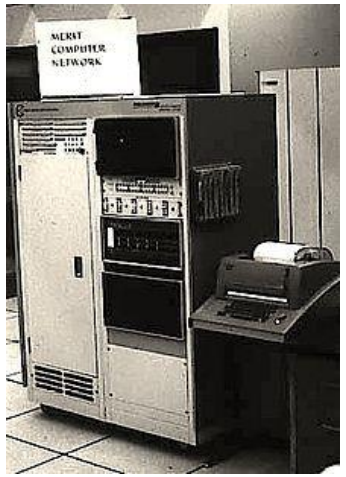
Introduction

- Séquencement de l’unité
- Historique
- Caractéristiques
- Domaines d’applications
- Différences avec Java
- Place du C dans le monde informatique
- Utilisation du C à l’ESIEE

Séquencement

- Total 30h (15hC + 15h TP)
- Alternance **cours**/TP : 1h TP, **2h C**, **2h C**, **2hC**, 2h TP, **2h C**, 2h TP, 2hTP**,** **2h C**, 2h TP, **2hC**, 2h TP, **2h C**, 2h TP, 2h TP, **1h C**
- Evaluation par examen final (questions de cours + exercices type TP)

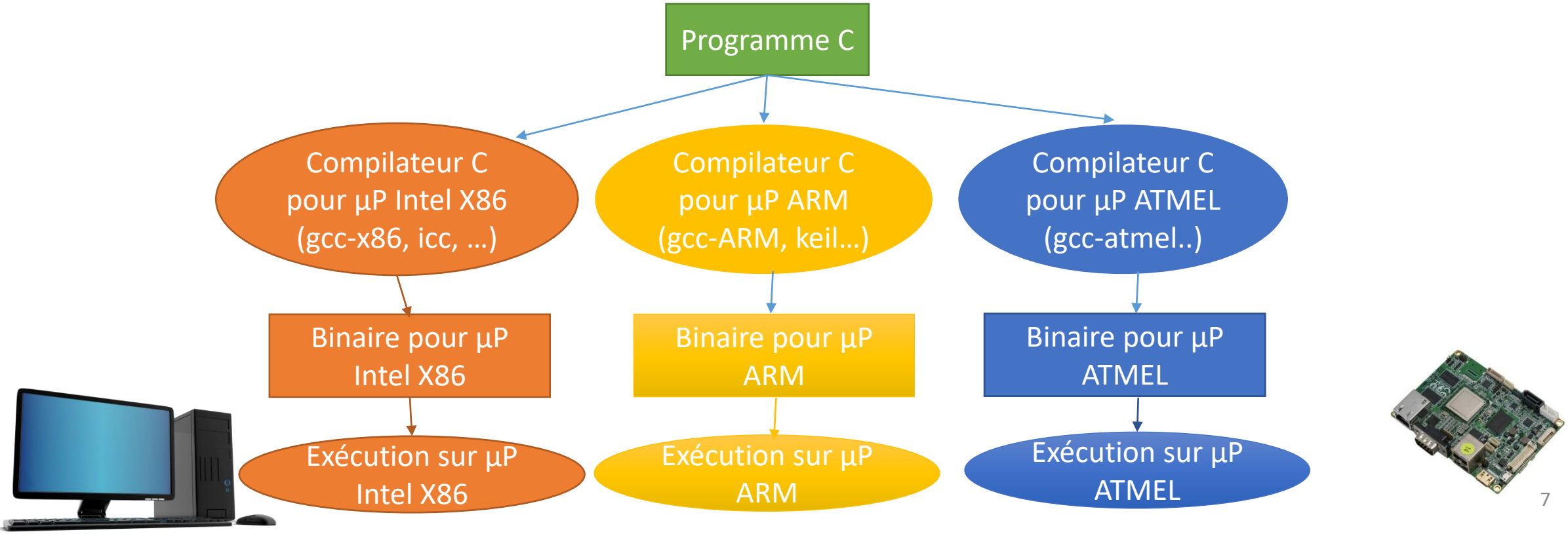
Historique du langage C



- Développé en 1972 dans les laboratoires Bell
  - En même temps que UNIX, sur PDP-11
  - Par Dennis Ritchie (1941-2011) et Ken Thompson (1943-) (lang. 'B')
  - Enrichi et popularisé par Brian Kernighan (1942-) (« The C programming Language » K&R)
- Normalisé en 1989 Norme ANSI C (C89) (la plus répandue) puis en 1999 norme ISO « C99 » et en 2011 « C11 ».

Caractéristiques

- Basé sur une chaîne de compilation : fichier source ➔ compilateur (…éditeur de liens..) ➔ binaire exécutable



Caractéristiques

- Plus haut niveau que l'assembleur, portable, peu restrictif : permet de rester proche du processeur (de la mémoire) mais en masquant l'utilisation des registres
- Langage de programmation impératif (≠ déclaratif) : on programme en indiquant des séquences d'opérations successives à exécuter par le processeur)
- (Faiblement typé)
- Le portage du compilateur est relativement simple

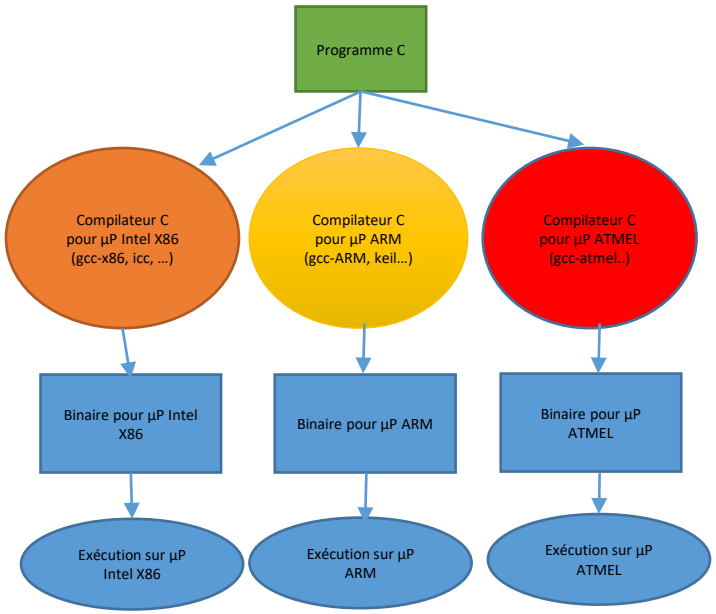
Domaines d'applications

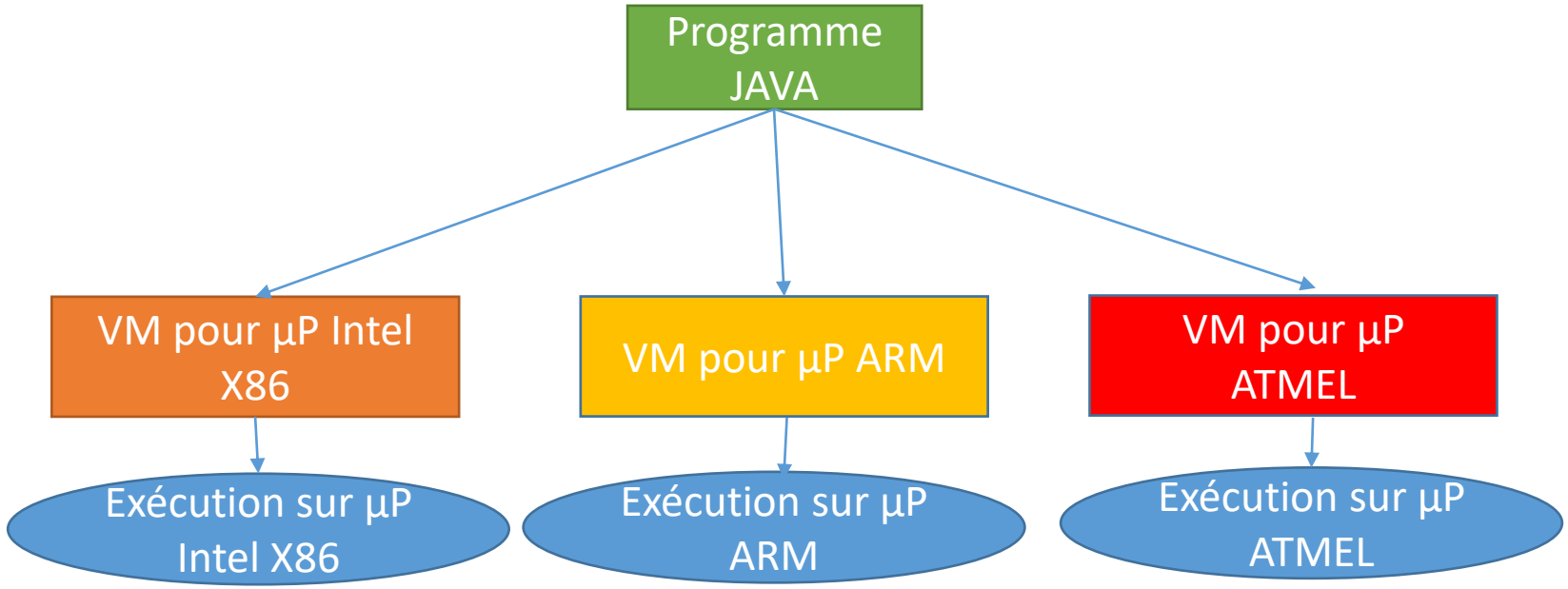
- Programmation système : OS (Windows, Linux, Machines Virtuelles…), OS temps réel + dev. des drivers de périphériques
- Les applications embarquées où les octets, les milliampères et le temps sont comptés (Objets connectés, objets autonomes, drones…)

  ➔programmation de microcontrôleurs (=µP embarqués)
- Quand besoin de performances (calcul haute performance) : pour être au plus près du système (maitrise des caches, pipeline, code assembleur généré).
- Besoin de maîtriser l'exécution (durée, ordre) : les applications critiques, temps réel.
- Quand les langages de plus haut niveau n'ont pas encore été portés sur un nouveau µP (ou ne le seront jamais : coprocesseur..)
- Cybersécurité : maitriser chaque maillon, injection de code ASM ou C

Différences avec Java : principe la VM

- La portabilité de Java vient de sa machine virtuelle (VM) qui est spécifique à chaque µP et à chaque OS
- C est un langage compilé, JAVA est interprété (idem pour Python...)





Différences avec Java : VM vs code natif

- La portabilité de Java vient de sa machine virtuelle (VM) qui est spécifique à chaque µP et à chaque OS, Java est interprété par la VM
- La VM prend des décisions « dynamiquement » (lors de l'exécution du programme) :
  - libération de la mémoire par le «ramasse miette » de la VM : non déterministe (quand, pendant combien de temps ?)
  - Pas de connaissance précise de ce que fait la VM (utilisation de la mémoire ?)
- C ne repose pas sur une VM : le compilateur produit du code assembleur directement exécuté par le processeur dans l'ordre indiqué dans ce code.

Différences avec Java : en C tout est permis…

- Comme C a un accès direct au µP, à la mémoire (pas de VM) c'est au programmeur de gérer la réservation (allocation & libération d'espace mémoire)...risque de bugs ! ➔ bien comprendre le fonctionnement
- Comme accès direct : risque de dépassement de tableaux (indice), d'écriture n'importe où dans la mémoire...

  ➔ Dégâts encore limités quand on est sur un système avec OS (Linux, Windows etc) mais très grand quand pas d'OS...

Une VM (ou un interpréteur) effectue plusieurs vérifications avant d'exécuter une instruction....plus de sûreté mais...plus de temps (et moins de contrôle)

car C est conçu pour coder des Operating System : il doit tout permettre sans restriction

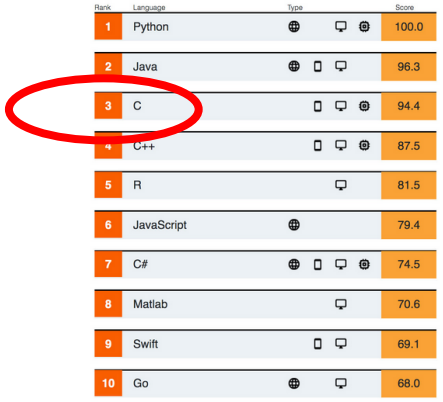
Différences avec Java : langage

- C n'est pas orienté objet (pas de classe, pas d'héritage)...
- Les structures de contrôle (if/else, switch, for, while, do-while...) de Java sont inspirées de C ➔ identiques.
  - ATTENTION : il faut déclarer les variables locales avant des les utiliser (C ANSI).
- Le nom des types mémoires sont parfois différents (byte = char...)
- Les tableaux, leur gestion, la mise en paramètres, l'allocation : grosses différences.
- Les fonctions ne sont pas rattachées au classes (méthode), différences sur passage des arguments...
- Les bibliothèques d'entrées-sorties (clavier, écran, fichier...) diffèrent.
- Pas d'exception pour gérer les erreurs comme en Java...

Différences avec Java : conclusion

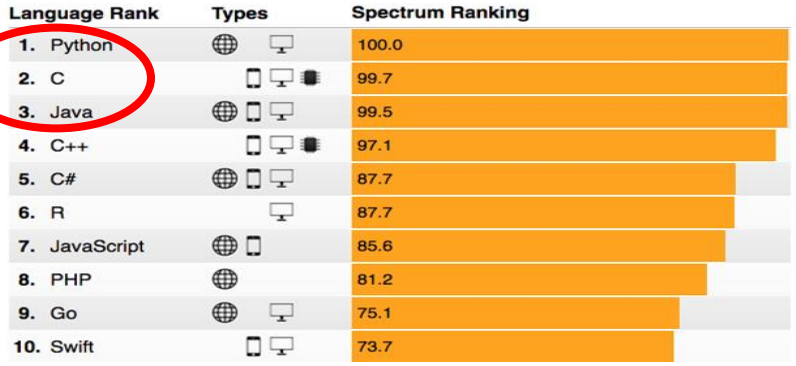
- Chaque langage est conçu pour un ou plusieurs domaines
- C est « bas niveau » : proche de la machine, adapté pour la conception d'OS, de driver, répond au besoin de maitrise de l'exécution : performances, déterminisme, occupation mémoire, consommation énergétique...
- JAVA est un langage de « haut niveau » bien adapté pour développer des applications de haut niveau que l'on souhaite portable, il repose sur de nombreuses bibliothèques.
- Il existe de nombreux autres langages de haut niveau, interprétés ou non mais C reste le langage de bas niveau le plus utilisé.

Le langage C : un besoin constant
ex. "IEEE : **The 2019 Top Programming Languages** "



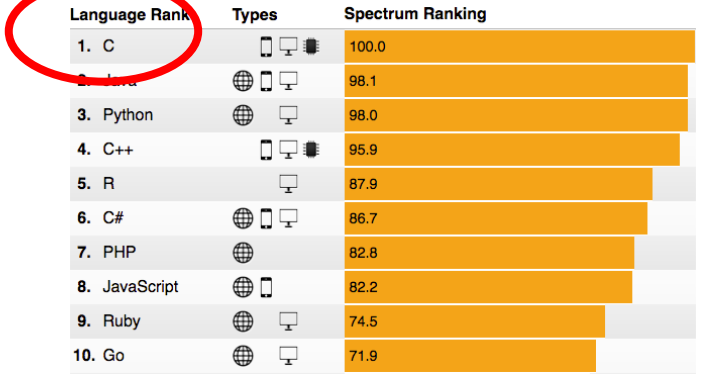
Rank	Language	Type	Score
1	Python		100.0
2	Java		96.3
3	C		94.4
4	C++		87.5
5	R		81.5
6	JavaScript		79.4
7	C#		74.5
8	Matlab		70.6
9	Swift		69.1
10	Go		68.0

2019



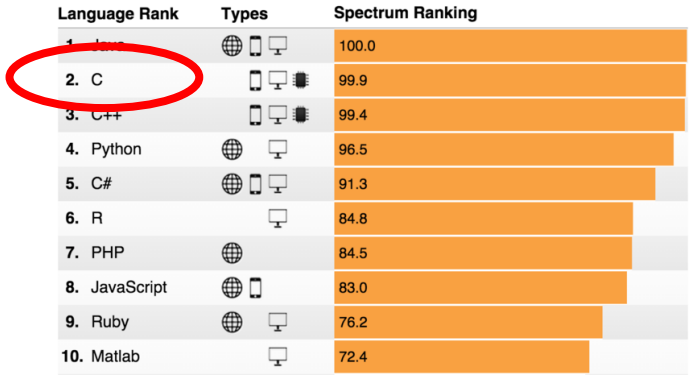
Language Rank	Types	Spectrum Ranking
1. Python		100.0
2. C		99.7
3. Java		99.5
4. C++		97.1
5. C#		87.7
6. R		87.7
7. JavaScript		85.6
8. PHP		81.2
9. Go		75.1
10. Swift		73.7

2017



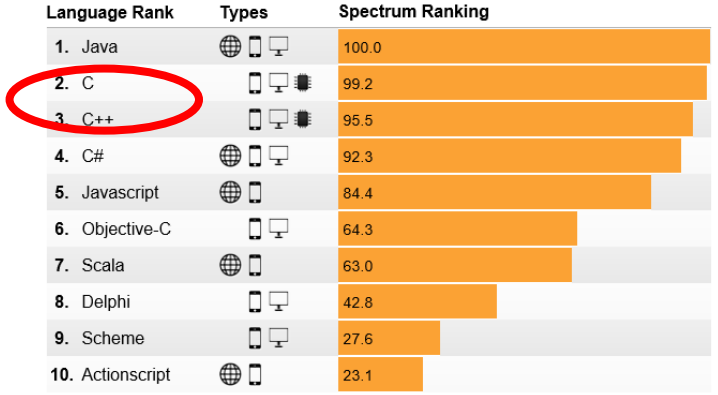
Language Rank	Types	Spectrum Ranking
1. C		100.0
2. Java		98.1
3. Python		98.0
4. C++		95.9
5. R		87.9
6. C#		86.7
7. PHP		82.8
8. JavaScript		82.2
9. Ruby		74.5
10. Go		71.9

2016



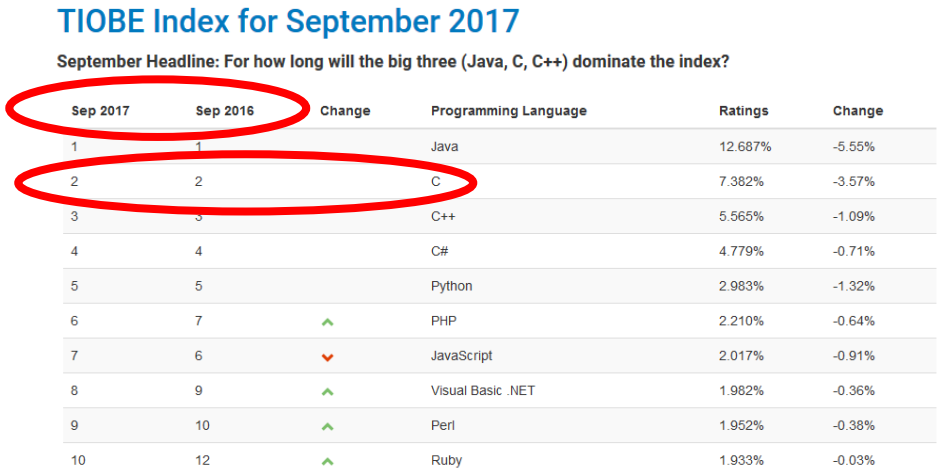
Language Rank	Types	Spectrum Ranking
1. Java		100.0
2. C		99.9
3. C++		99.4
4. Python		96.5
5. C#		91.3
6. R		84.8
7. PHP		84.5
8. JavaScript		83.0
9. Ruby		76.2
10. Matlab		72.4

2015



Language Rank	Types	Spectrum Ranking
1. Java		100.0
2. C		99.2
3. C++		95.5
4. C#		92.3
5. Javascript		84.4
6. Objective-C		64.3
7. Scala		63.0
8. Delphi		42.8
9. Scheme		27.6
10. Actionscript		23.1

2014



TIOBE Index for September 2017

September Headline: For how long will the big three (Java, C, C++) dominate the index?

Sep 2017	Sep 2016	Change	Programming Language	Ratings	Change
1	1		Java	12.687%	-5.55%
2	2		C	7.382%	-3.57%
3	3		C++	5.565%	-1.09%
4	4		C#	4.779%	-0.71%
5	5		Python	2.983%	-1.32%
6	7	^	PHP	2.210%	-0.64%
7	6	v	JavaScript	2.017%	-0.91%
8	9	^	Visual Basic .NET	1.982%	-0.36%
9	10	^	Perl	1.952%	-0.38%
10	12	^	Ruby	1.933%	-0.03%

https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages

ou google « top programming language »

C et unités + filières ESIEE

- En E3 :
  - Systèmes d'exploitation – programmation système
  - Programmation des microcontrôleurs
- Filière informatique (HPC, architecture, optimisation, prog. système avancée, GPU...)
- Filière Cybersécurité (failles, attaques bas niveau, injections...)
- Filière Système Embarquée (OS temps réel, driver, matériel...)
- Filière Santé/Energie/Environnement (interface avec HW, minimisation de la conso., autonomie...)

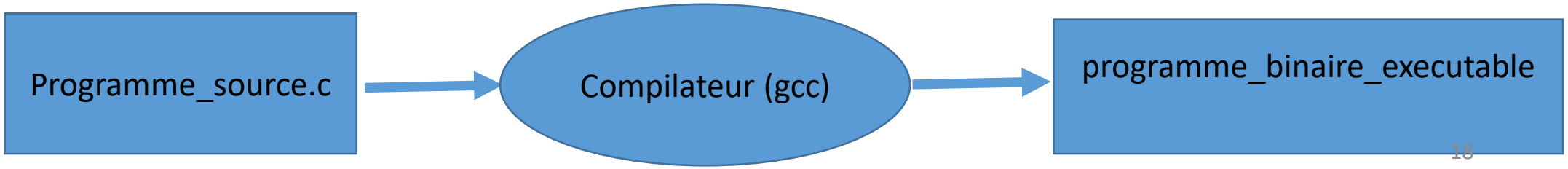
II- Premier contact

Premier programme

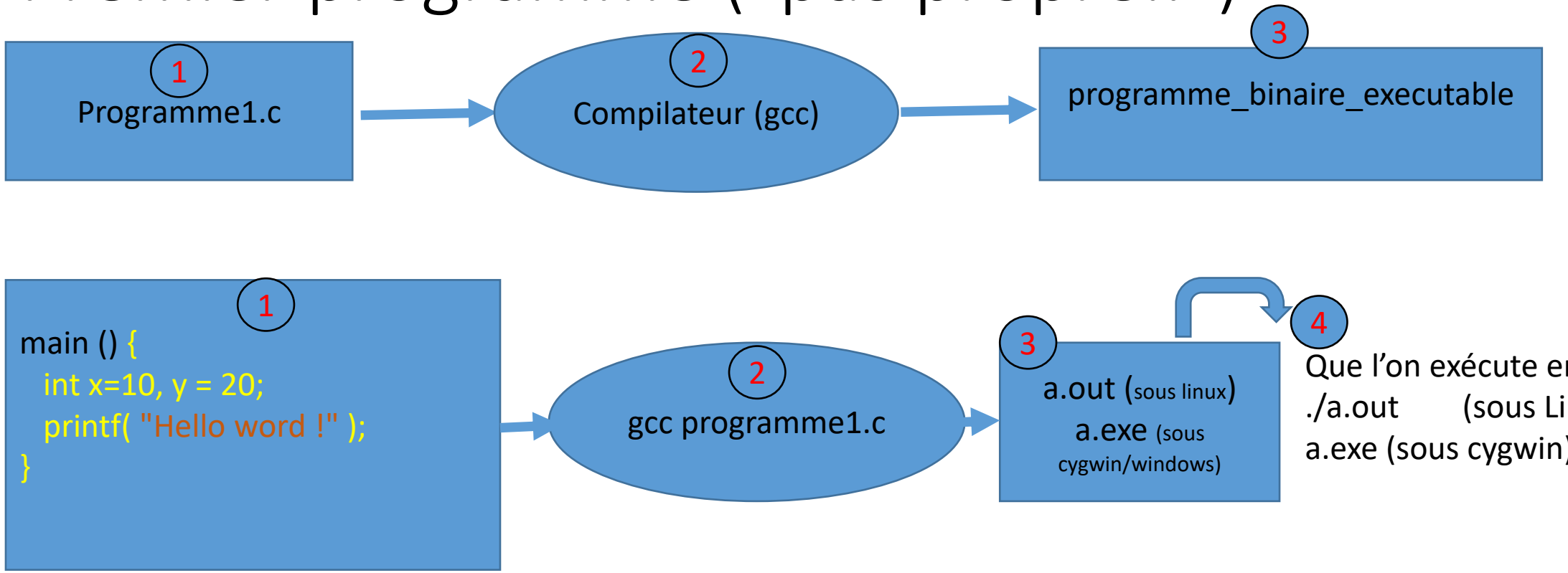
Premier programme

- Editeur de texte simple (sans mise en forme) : notepad, notepad++, nedit, gedit,kedit, kate etc ≠ un traitement de texte car pour la mise en forme (couleurs, taille etc…) insère des caractères particuliers invisibles
  - Le fichier produit=fichier source, doit avoir l'extension .c (ou .h que nous verrons plus tard)
- Un compilateur qui transforme le fichier source en binaire exécutable (en réalité le compilateur appel un ensemble d'outils) : gcc, icc,…

  En fonction du compilateur utilisé :
  - Ce binaire est exécutable sur la même machine (« compilation native »)
  - Ce binaire est exécutable sur une autre machine : « cross compilation »



Premier programme ("pas propre..")



- main = fonction indispensable qui est appelée lorsque l'on exécute le programme. Ici on déclare le code de la fonction main.
- On déclare le code d'une fonction entre accolades = corps de la fonctions.
- Dans main on utilise printf qui est une fonction existante : les arguments des fonc. sont entre parenthèses.
- Chaque : instruction, déclaration, appel de fonc. doit être suivi d'un point-virgule.

Premier programme (version "propre")

- En compilant ce (mauvais!) programme le compilateur génère quelques avertissements (warning Cf. slides suivant).
- Il faut en effet donner plus d'information dans le code source :
  - indiquer où trouver le prototype (=signature) des fonctions utilisées (ici printf)
    - on ajoute la directive #include <Nom_fichier.h> ⇔ copier/coller du fichier au début de ce programme. Nom_fichier.h contient le prototype de printf
  - indiquer que main retourne une valeur (int) + retourner cette valeur (return 0) (sert à indiquer au terminal si le prog. s'est correctement terminé).

```
#include <stdio.h>
int main () {
   int x=10, y = 20;
   printf( "Hello word !");
   return 0;
}
```

Premier programme

- L'argument –Wall (=Warning all) de gcc : permet d'afficher tous les avertissements (warning) de gcc
  - **INDISPENSABLE en C** ne jamais s'en priver !
- Exemple sur la première version du programme (programme1.c) :

  pc5103a: **gcc programme1.c -Wall**

  1erProg.c:1:1: warning: return type defaults to ‘int’ [-Wreturn-type]

  1erProg.c: In function ‘main’:

  1erProg.c:3:2: warning: implicit declaration of function ‘printf’ [-Wimplicit-function-declaration]

  1erProg.c:3:2: warning: incompatible implicit declaration of built-in function ‘printf’ [enabled by default]

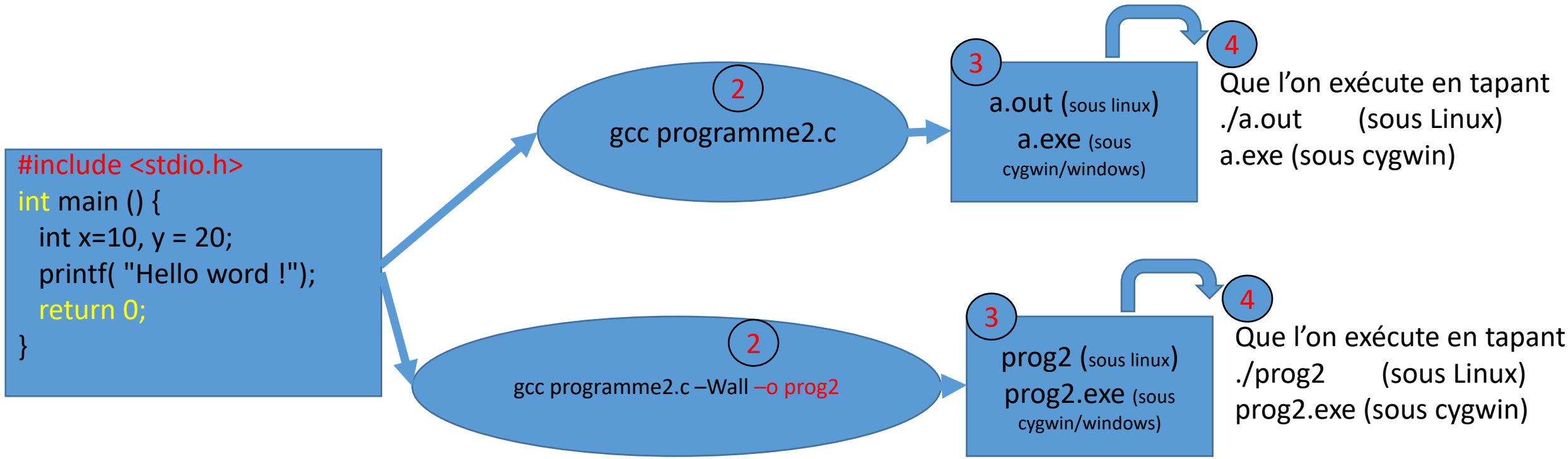
  1erProg.c:2:12: warning: unused variable ‘y’ [-Wunused-variable]

  1erProg.c:2:6: warning: unused variable ‘x’ [-Wunused-variable]

  1erProg.c:5:1: warning: control reaches end of non-void function [-Wreturn-type]

Premier programme

- L'argument –o de gcc : permet de spécifier le nom du programme à créer



Rappel des commandes Unix/Linux

- *ls* et ses options (*ls –a* par exemple)
- *mkdir répertoire* (par exemple : *mkdir TP_IGI2101*)
- *cd répertoire* (par exemple *cd TP_IGI2101*)
- *cd ..*
- *cd ~*
- *cd .*
- *cd ~/TP_IGI2101/TP1*
- *pwd*
- Rôle de la variable PATH (affichable avec *echo $PATH*)
- La commande *man* , par exemple *man ls*, *man cd*, *man gcc (lire les crochets comme des arguments optionnels)*

Composition d'un programme

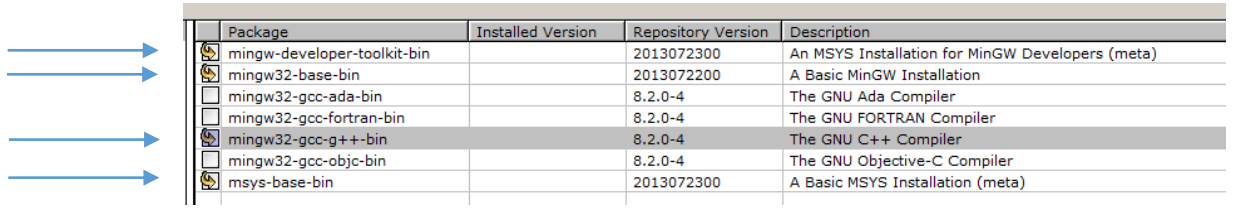
- directives au préprocesseur (#include<NomFich.h> et #include "NomFich.h")
  - le préprocesseur est exécuté en 1er lors de la compilation : il fait des modif. textuelles (copier/coller, chercher/remplacer) dans le fichier source (Cf. plus tard : #define, #ifdef etc),
  - possibilité de voir le source après passage dans préprocesseur avec argument –E
- de déclarations de fonctions (au moins la fonction main !),
- de déclarations de variables (regroupées au début de chaque déclaration en C ANSI),
- d'instructions (affectations, opérateurs, structures de ctrl), appel de fonction,
- **de commentaires** ! /* bloc multi lignes de commentaires */ et // (depuis C99, vient du C++).

Structure d'un programme "mono fichier"

- Directives du préprocesseur
- Déclaration des fonctions
- Déclaration de la fonction main () {
  - Déclaration des variables locales
  - Instructions
  - return 0;

  }
- Possible ici aussi : déclaration des fonctions (mais alors ajouter leur prototype avant utilisation : i.e par exemple avant le main.

Pour programmer chez vous :



	Package	Installed Version	Repository Version	Description
☑	mingw-developer-toolkit-bin		2013072300	An MSYS Installation for MinGW Developers (meta)
☑	mingw32-base-bin		2013072200	A Basic MinGW Installation
☐	mingw32-gcc-ada-bin		8.2.0-4	The GNU Ada Compiler
☐	mingw32-gcc-fortran-bin		8.2.0-4	The GNU FORTRAN Compiler
☑	mingw32-gcc-g++-bin		8.2.0-4	The GNU C++ Compiler
☐	mingw32-gcc-objc-bin		8.2.0-4	The GNU Objective-C Compiler
☑	msys-base-bin		2013072300	A Basic MSYS Installation (meta)

- Plusieurs possibilités :
  - Sous Windows :
    - Utiliser Mingw (http://www.mingw.org), télécharger "*Download mingw-get-setup.exe*" et sélectionner tous les packages sauf Ada, Fortran, Objective-C (voir ma page web/explications sur Blackboard)
    - Installer Cygwin
    - Fabriquer une clef USB bootable sous linux (voir ma page web)
    - Attention je ne recommande pas Code::Blocks pour débuter (masque certaine chose)
  - Sous Mac OS :
    - Il suffit d'installer xcode
  - Installer Linux sur le disque dur (dual boot : boot Linux et boot Windows)
  - Travailler dans le cloud : exemple http://ideone.com

Règles importantes

1. **Toujours indenter** ! (tab ou espace pour chaque ligne suivant une accolade ouvrante)

2. Toujours compiler avec l'option **–Wall**

3. En C : pas de déclaration de variables ailleurs qu'au début du bloc de code (pas dans la boucle for!)

III- Les types et structures de données

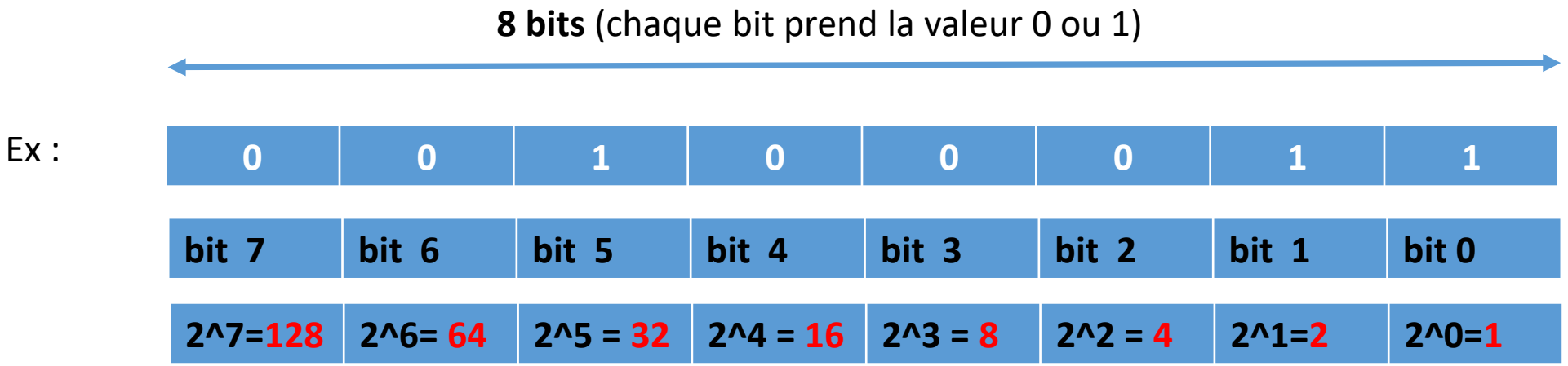
T.Grandpierre

Types et structures de données

- Types entiers
  - char / unsigned char
  - affichage en hexa, opérateur sizeof, codage ASCII
  - short, endianness
  - int
- Type flottants
  - Limites du codage en virgule fixe
  - Codage IEEE 754 (float, double)
  - Avantage inconvénients : FPU…
- Tableaux
  - 1D, déclaration, initialisation, allocation mémoire, erreurs fréquentes
  - 1D Chaîne de caractères, bibliothèque string.h
  - 2D
- Structures
- Les pointeurs (1ère partie)

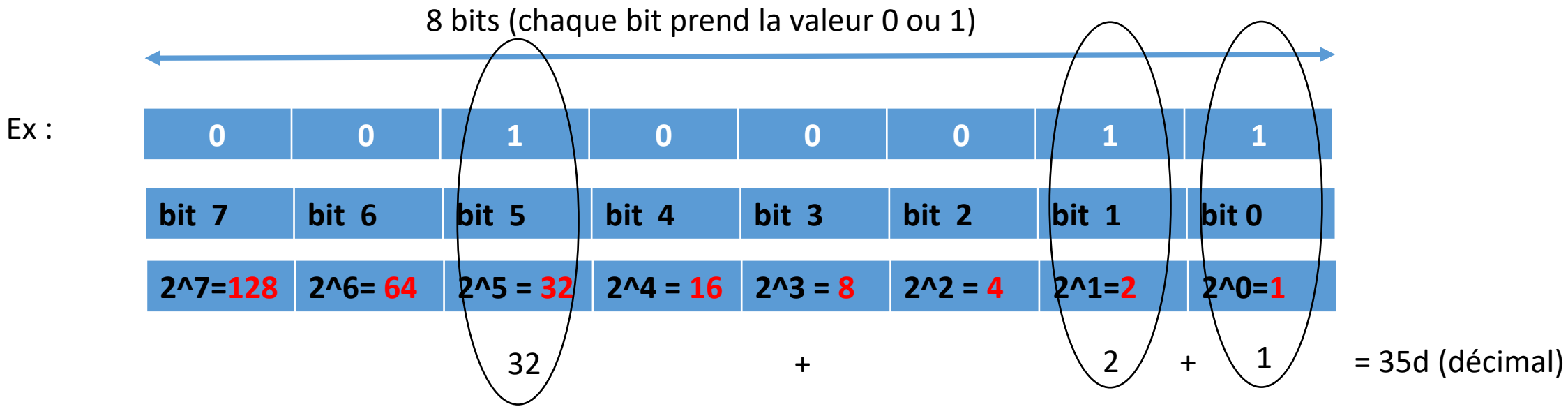
Les types entiers : 8 bits (=1 octet)

- **char** : 8bits , codé en complément à 2, valeur entre ……. et …….
- **unsigned char,** valeur entre ……..et……..



- **Attention en Java : byte**

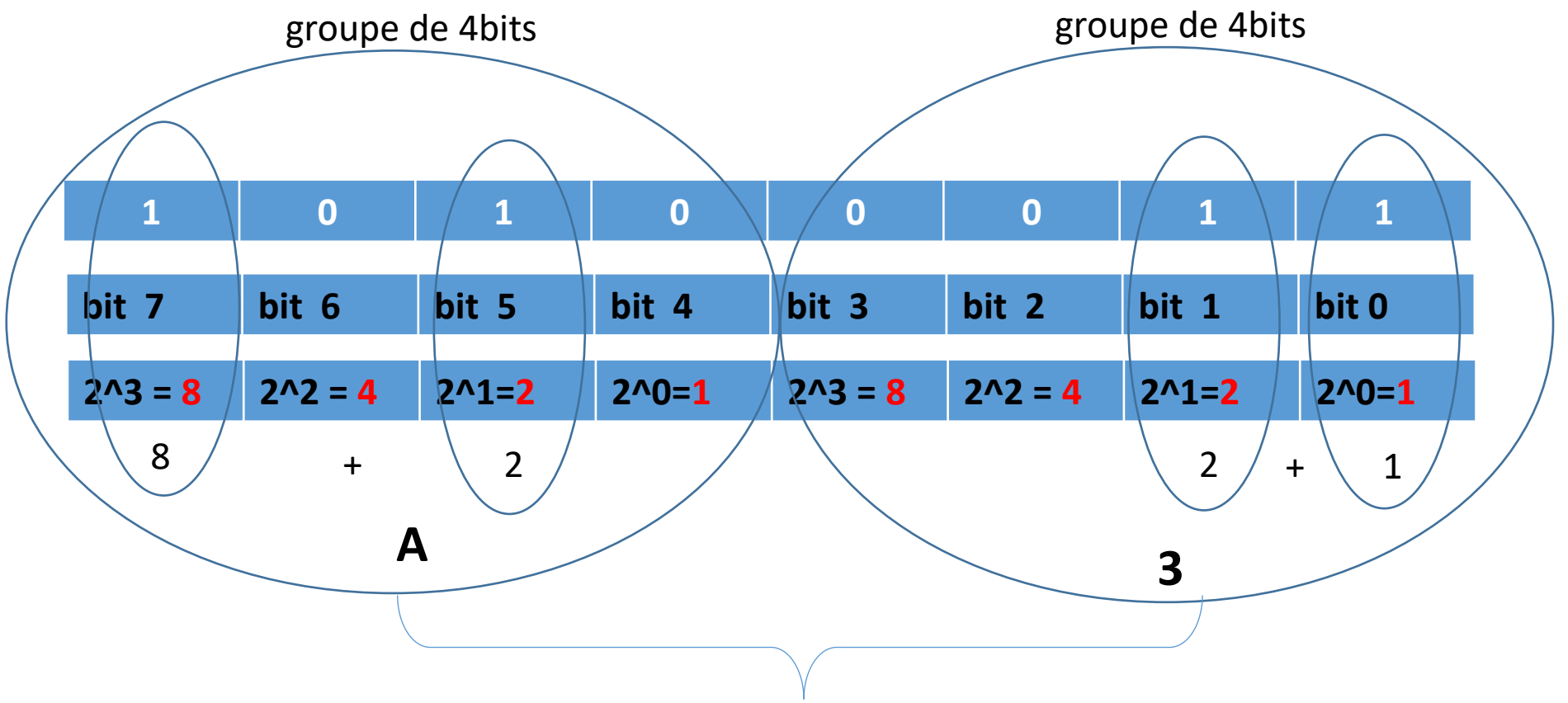
Les types entiers : 8 bits rappel binaire



0010 0011b (binaire) = 35d (décimal)

Les types entiers : 8 bits, rappel hexadécimal

Ex :



0xA3 = 1010 0011b = 163d

On **écrira unsigned char x = 0xA3;** ou **unsigned char x = 0XA3; ⇔ unsigned char x=163;**

0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Les types entiers 8 bits : affichage

```
#include <stdio.h>
int main () {
   char y=10;
   unsigned char z = 163;
   printf("y=%d , z=%d", y, z);
   return 0;
}
```

Affichera :

y=10, z=163

on remplace

```
printf("y= %x, z= %X'', y, z);
```

Affichera :

y=a, z=A3

N.B. on pouvait aussi déclarer:

```
char y=0xA, z=0xA3;
printf("y=%d , z=%d", y, z);
```

Affichera aussi :

y=10, z=163

Les données sont stockées en binaire dans la machine, mais on peut les affecter et les afficher dans la base (décimale, hexadécimale...) de son choix !

Les types entiers 8 bits : opérateur sizeof( )

```
int main () {
   char y=10;
   unsigned char z = 20;
   printf("y=%d , z=%d", y,z);
   return 0;
}
```

Pour connaitre la taille d'une variable : sizeof(NomVariable) ou sizeof(NomType)

Exemple :

```
printf("taille de y= %d octet", sizeof(y));
printf("taille de z= %d octet", sizeof(z));
```

Affichera :

taille de y=1 octet

taille de z=1 octet

N.B ceci est equivalent à écrire :

```
char y=10, t=0;
unsigned char z = 20;
t=sizeof(y);
printf("taille de y= %d octet", t);
```

Attention : sizeof( ) est remplacé par une *constante* au moment de la compilation

Char et code ASCII : rappel code ASCII

Encode chaque caractère par un numéro standardisé

(ASCII *American Standard Code for Information Interchange)*

'0' ⇔ 48d
....donc 9 = 48+9 = 57
'A' ⇔ 65d
...donc Z = 65+25 = 90 (A=65+0=65)
'a' ⇔ 61d

Jamais besoin de les retenir (Cf. slide suivant)

Non affichables : controle

Caractères affichables

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

The ASCII code

American Standard Code for Information Interchange

ASCII control characters

DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de linea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift Out)
15	0Fh	SI	(shift In)
16	10h	DLE	(data link escape)
17	11h	DC1	(device control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(device control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	20h	DEL	(delete)

ASCII printable characters

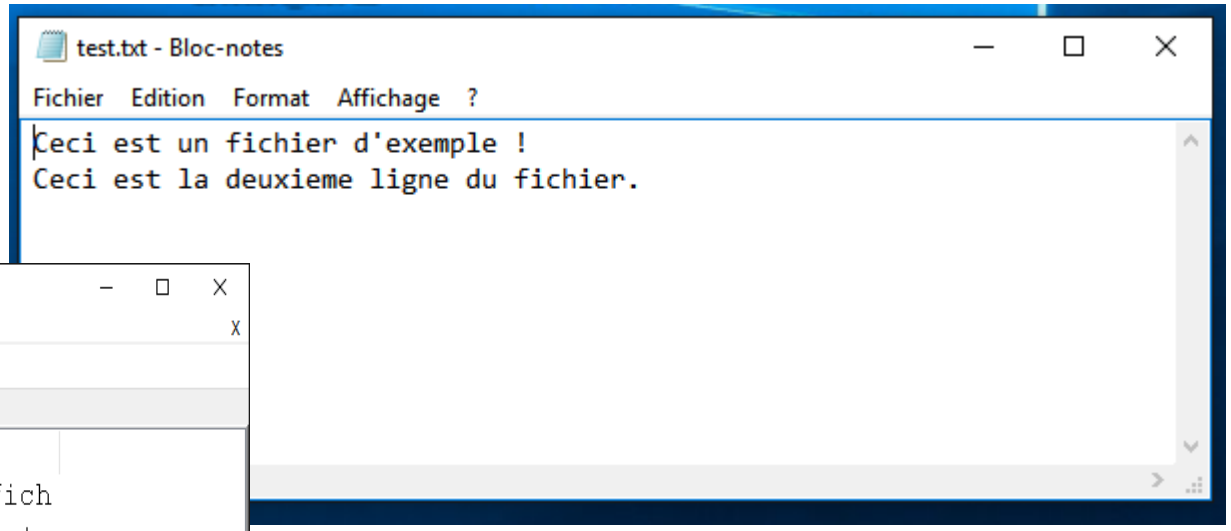
DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

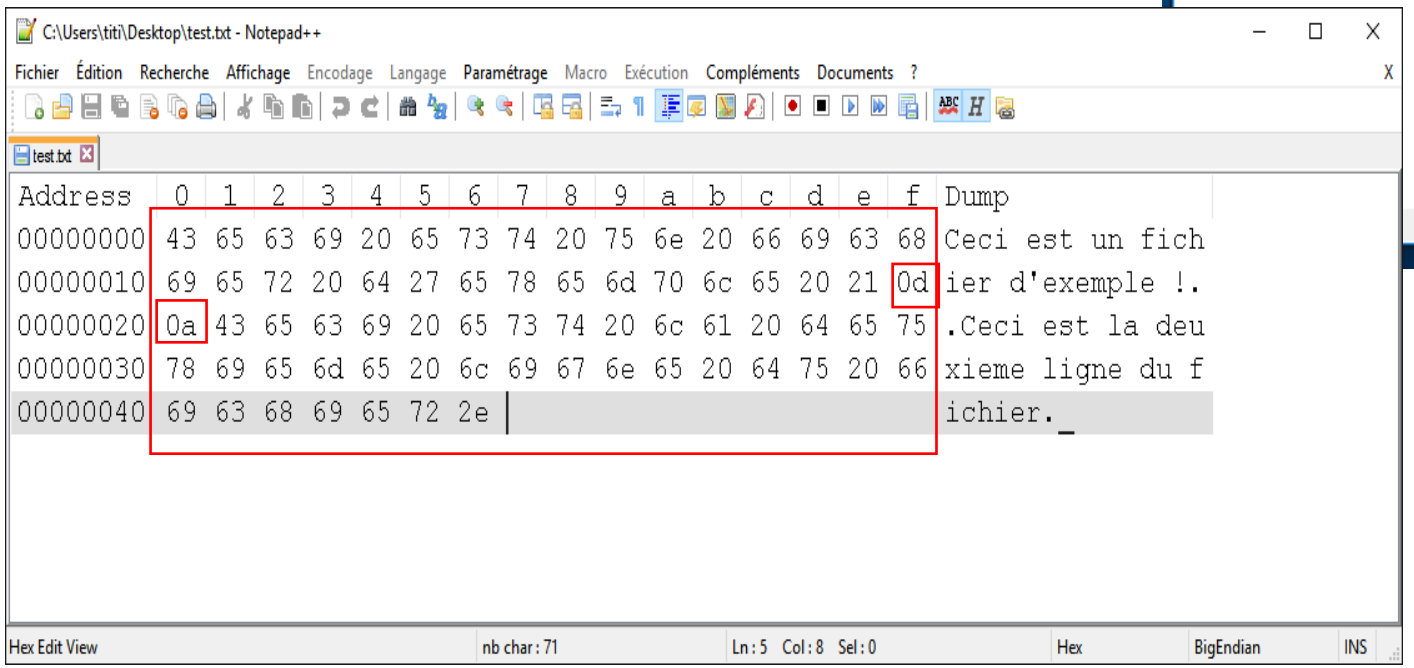
theASCIIcode.com.ar

Extended ASCII characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	└	224	E0h	Ó
129	81h	ü	161	A1h	í	193	C1h	┴	225	E1h	ß
130	82h	é	162	A2h	ó	194	C2h	┬	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	├	227	E3h	Ò
132	84h	ä	164	A4h	ñ	196	C4h	─	228	E4h	õ
133	85h	à	165	A5h	Ñ	197	C5h	┼	229	E5h	Õ
134	86h	å	166	A6h	ª	198	C6h	ã	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ã	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	╚	232	E8h	Þ
137	89h	ë	169	A9h	®	201	C9h	╔	233	E9h	Ú
138	8Ah	è	170	AAh	¬	202	CAh	╩	234	EAh	Û
139	8Bh	ï	171	ABh	½	203	CBh	╦	235	EBh	Ù
140	8Ch	î	172	ACh	¼	204	CCh	╠	236	ECh	ý
141	8Dh	ì	173	ADh	¡	205	CDh	═	237	EDh	Ý
142	8Eh	Ä	174	AEh	«	206	CEh	╬	238	EEh	¯
143	8Fh	Å	175	AFh	»	207	CFh	¤	239	EFh	´
144	90h	É	176	B0h	░	208	D0h	ð	240	F0h	≡
145	91h	æ	177	B1h	▒	209	D1h	Ð	241	F1h	±
146	92h	Æ	178	B2h	▓	210	D2h	Ê	242	F2h	‗
147	93h	ô	179	B3h	│	211	D3h	Ë	243	F3h	¾
148	94h	ò	180	B4h	┤	212	D4h	È	244	F4h	¶
149	95h	ò	181	B5h	Á	213	D5h	ı	245	F5h	§
150	96h	û	182	B6h	Â	214	D6h	Í	246	F6h	÷
151	97h	ù	183	B7h	À	215	D7h	Î	247	F7h	¸
152	98h	ÿ	184	B8h	©	216	D8h	Ï	248	F8h	°
153	99h	Ö	185	B9h	╣	217	D9h	┘	249	F9h	¨
154	9Ah	Ü	186	BAh	║	218	DAh	┌	250	FAh	·
155	9Bh	ø	187	BBh	╗	219	DBh	█	251	FBh	¹
156	9Ch	£	188	BCh	╝	220	DCh	▄	252	FCh	³
157	9Dh	Ø	189	BDh	¢	221	DDh	¦	253	FDh	²
158	9Eh	×	190	BEh	¥	222	DEh	Ì	254	FEh	■
159	9Fh	ƒ	191	BFh	┐	223	DFh	▀	255	FFh	

Fichier texte : contenu ASCII



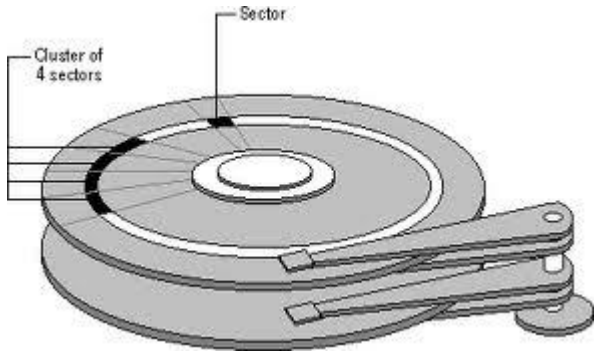


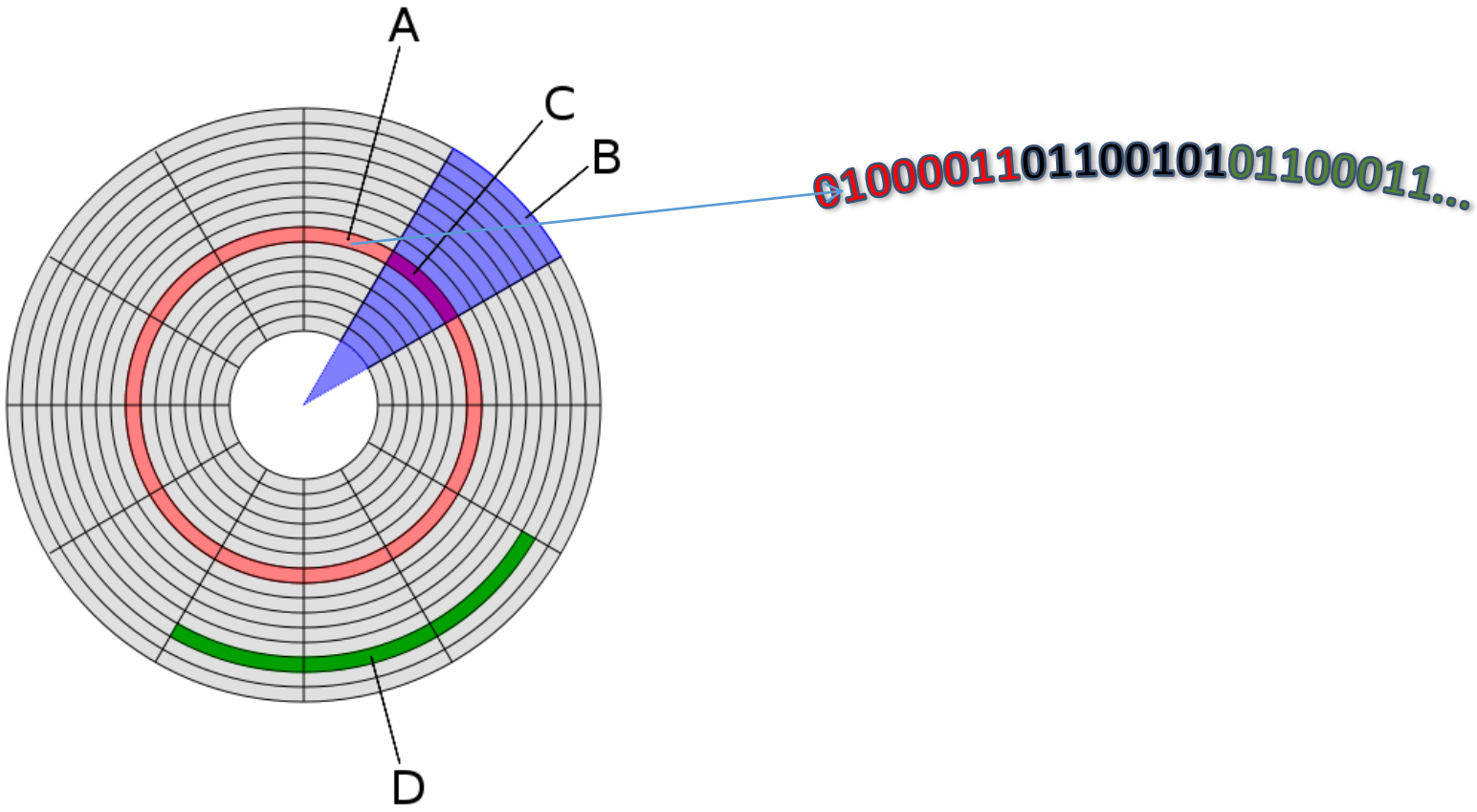
0x43='C'=01000011
0x65='e' =01100101
0x63='c' = 01100011
0x69='i' = 01101001
0x20= ' '= 00100000
0x65='e'= 01100101
etc

fichier sur le disque dur

Fichier texte : contenu ASCII

0x43='C'=01000011
0x65='e' =01100101
0x63='c' = 01100011
0x69='i' = 01101001
0x20= ' '= 00100000
0x65='e'= 01100101
etc





A Track
B Geometrical sector
C Track sector
D Cluster

Char et code ASCII

- On peut affecter la valeur du code ASCII d'un caractère dans un char :
  - unsigned char x = 'A'; ⇔ unsigned char x = 65;

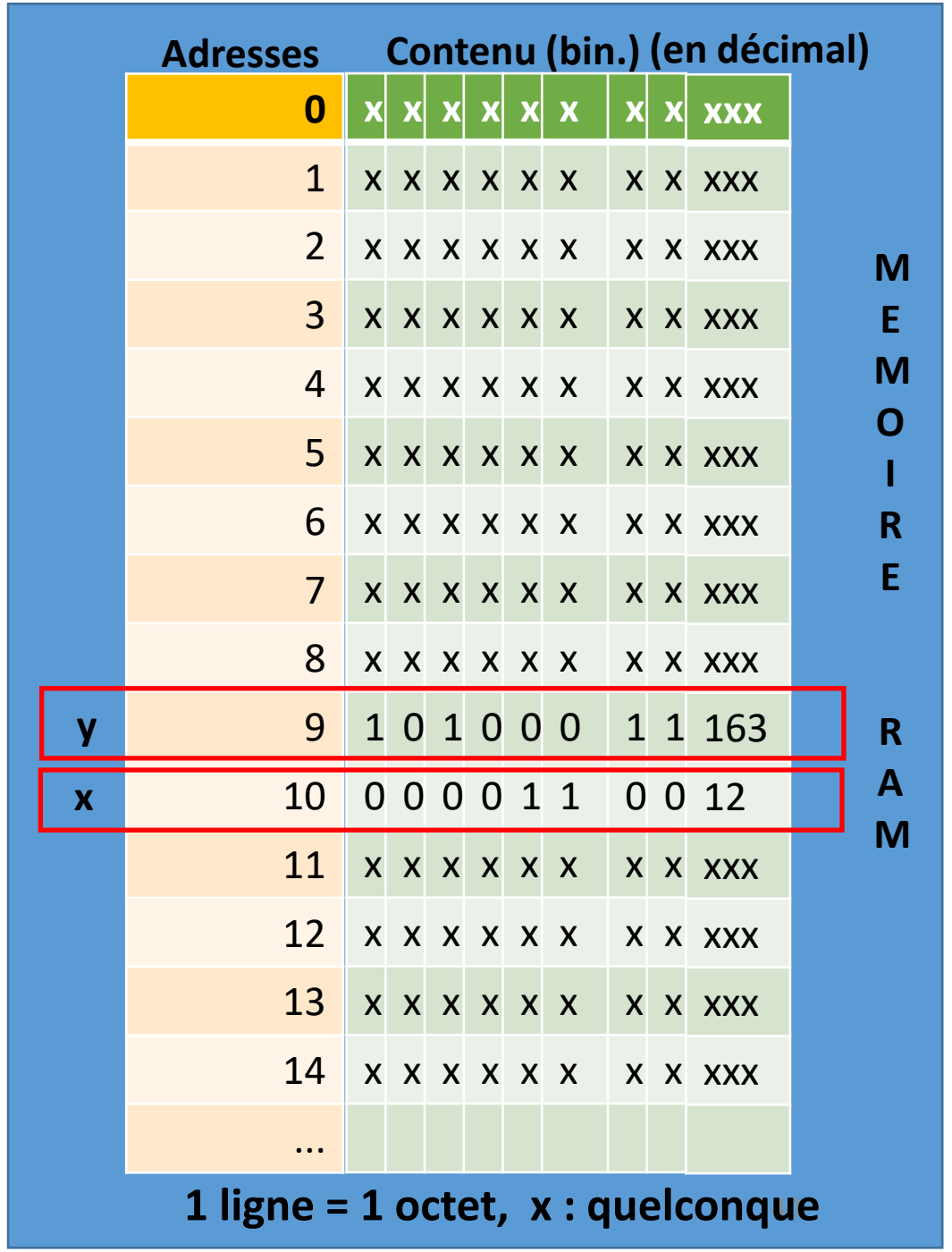
  - On peut afficher le char sous forme décimal, hexadécimal ou le caractère correspondant :
    unsigned char var='A';
    printf("var = %d ", var); ➔ affichera var = 65
    printf("var = %x ", var); ➔ affichera var = 41
    printf("var = %c ", var); ➔ affichera var = A

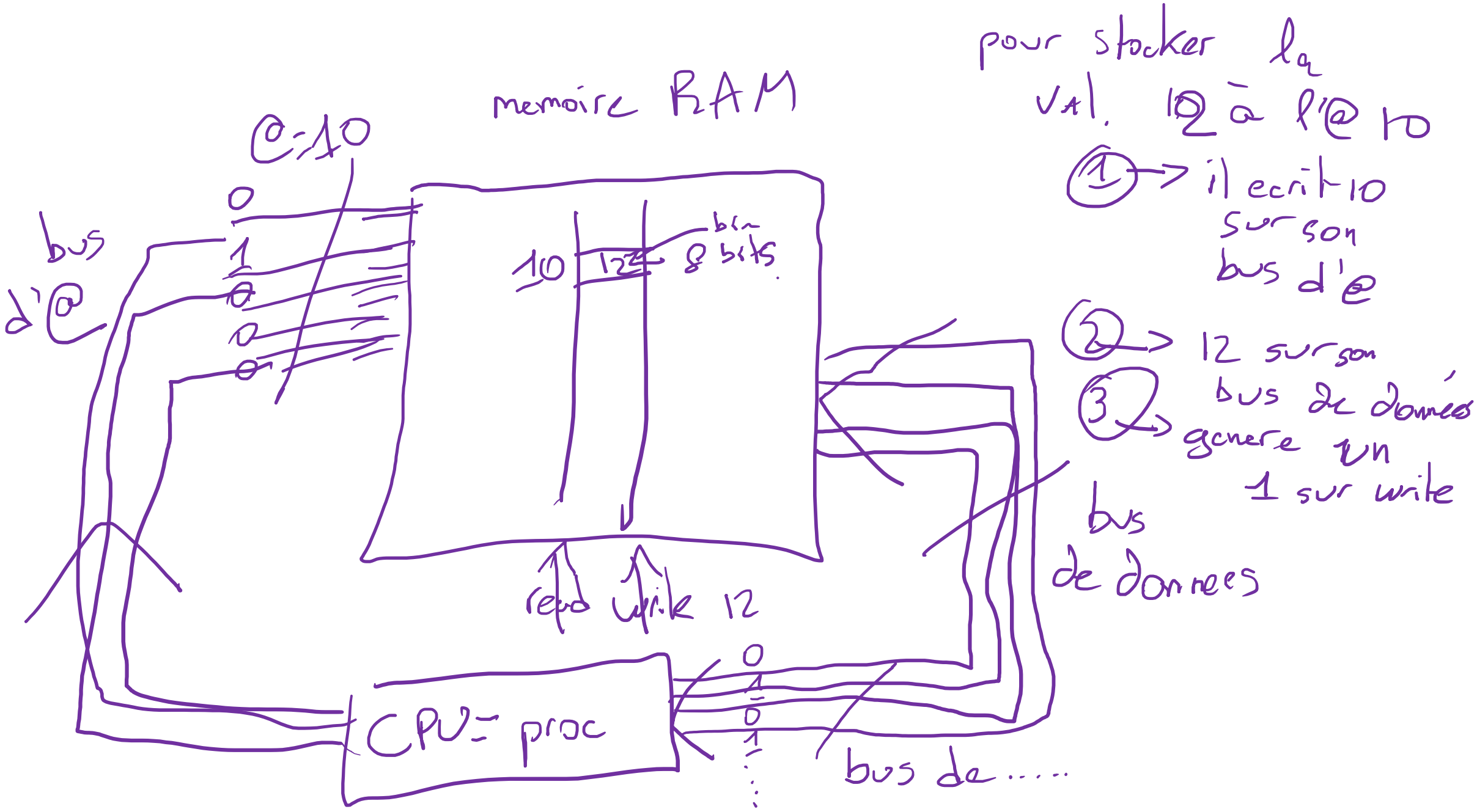
    %c pour afficher sous forme de caractère
  - L'arithmétique est donc inchangée : (puisque en mémoire var est un nombre !!)
    var = var+3; ⇔ var = 'A'+3;
    printf("var = %d ", var); ➔ affichera var = 68
    printf("var = %x ", var); ➔ affichera var = 44
    printf("var = %c ", var); ➔ affichera var = D

Allocation mémoire : pile

```
#include <stdio.h>
int main () {
   char x=12;
   unsigned char y=163;
   printf("x=%d , y=%d", x, y);
   return 0;
}
```

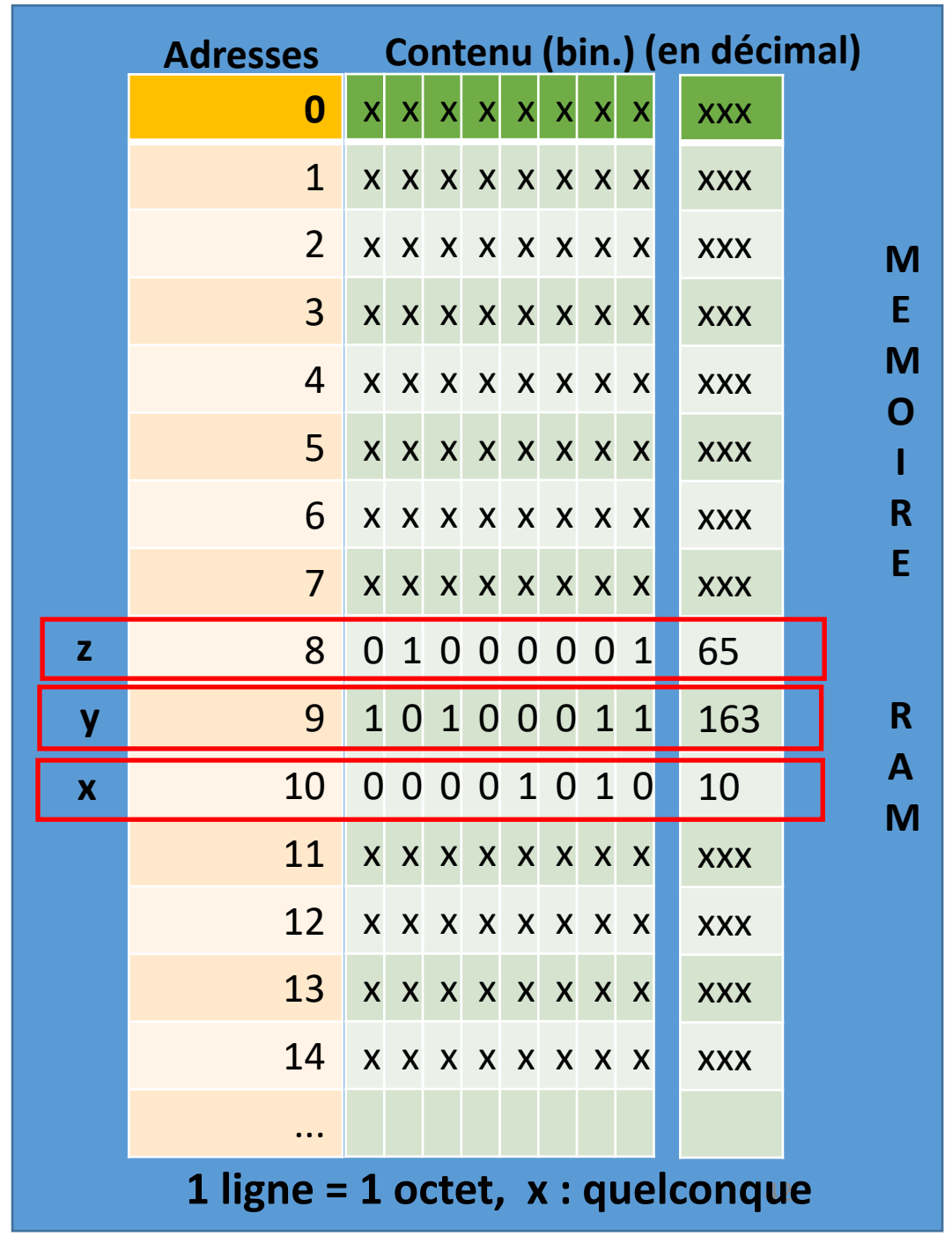
➔ variables locales du main (=fonctions) stockées à des adresses décroissante (on parle de "pile")




pour stocker la val. 12 à l'@ 10
1 → il écrit 10 sur son bus d'@
2 → 12 sur son bus de données
3 → genere un 1 sur write
mémoire RAM
@=10
bus d'@
0
1
0
0
0
10
12
bin 8 bits
read
write
12
bus de données
CPU = proc
0
1
0
1
bus de ....

Allocation mémoire : @

```
#include <stdio.h>
int main () {
   char x=10;
   unsigned char y=163;
   char z='A';
   printf("x=%d , y=%d", x, y);
   return 0;
}
```



Allocation mémoire : aff. @

opérateur unaire & : retourne l'adresse

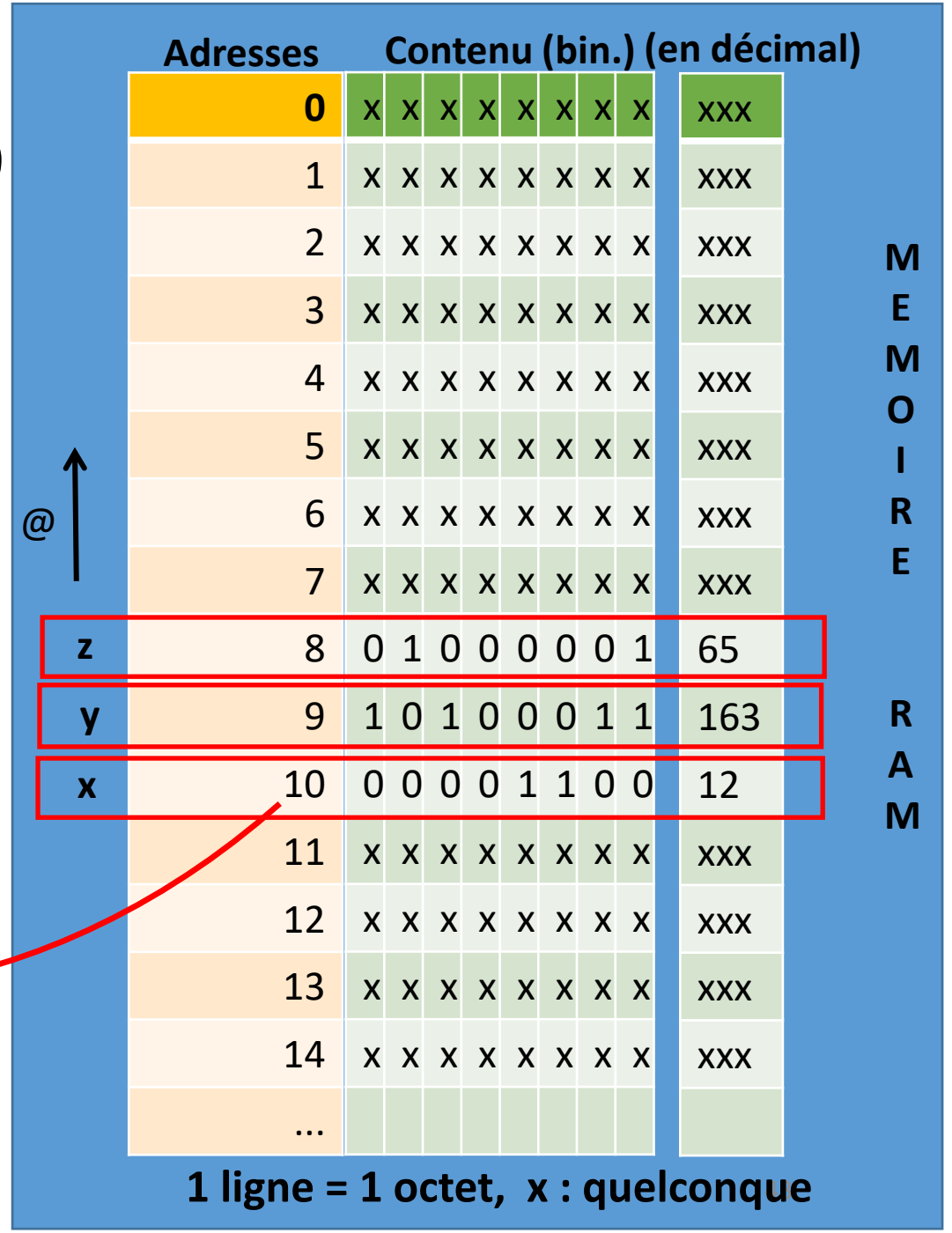
```
#include <stdio.h>
int main () {
   char x=12;
    unsigned char y=163;
    char z='A';
    printf("x=%d , y=%d, ", x, y);
    printf("adresse de z= %p", &z);
    return 0;
}
```

Affichera :

x=12, y = 163, adresse de z = 0x8 (car %p en hexa)

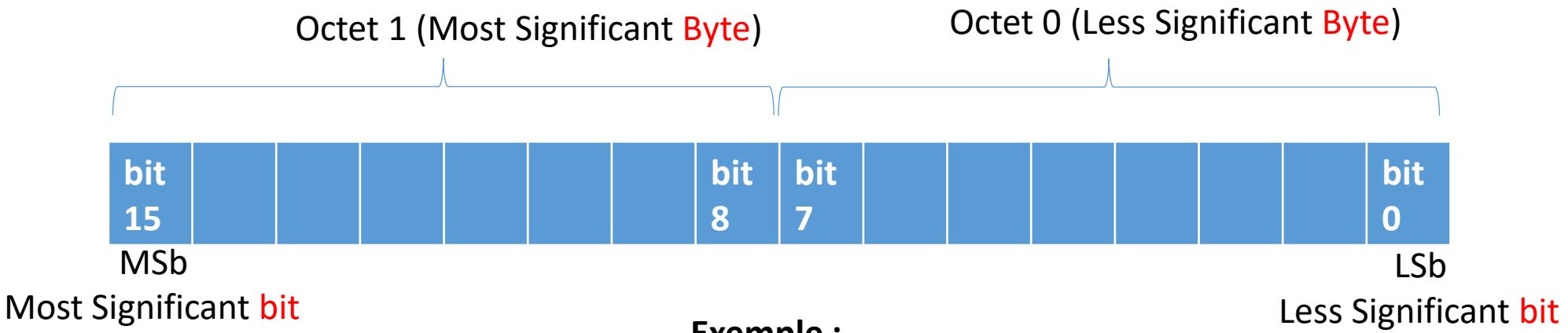
On ne peut pas stocker une adresse dans un char

(voir chapitre "pointeur"...)



Les types entiers : 16 bits (2 octets) ≠char java

- **short** : 16bits , codé en complément à 2, valeur entre …… et ……
- **unsigned short,** valeur entre …… et ……

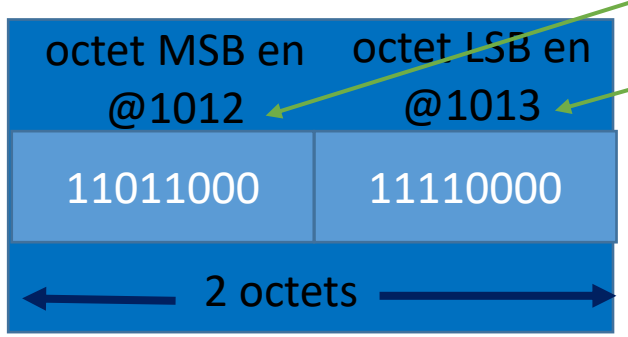


Exemple :

```
short a = -10000;
unsigned short b= 12000;
printf("a= %d b=%d", a,b);
sizeof (a) = ….. octets
sizeof (b) = ….. octets
```

Les types entiers : short

```
#include <stdio.h>
int main () {
   short a = -10000;
   unsigned short b= 12000;
   printf("a= %d b=%d", a,b);
   return 0;
}
```



	Adresses	Contenu (bin.) (en décimal)	
	1002	x x x x x x x x	xxx
	1003	x x x x x x x x	xxx
	1004	x x x x x x x x	xxx
	1005	x x x x x x x x	xxx
	1006	x x x x x x x x	xxx
	1007	x x x x x x x x	xxx
	1008	x x x x x x x x	xxx
b	1010	0 0 1 0 1 1 1 0	(MSB) 12000
b	1011	1 1 1 0 0 0 0 0	(LSB)
a	1012	1 1 0 1 1 0 0 0	(MSB) -10000
a	1013	1 1 1 1 0 0 0 0	(LSB)
	1014	x x x x x x x x	xxx
	1015	x x x x x x x x	xxx
	1016	x x x x x x x x	xxx
	1016	x x x x x x x x	xxx
	...		

2oct. (1010–1011) 2oct. (1012–1013)

MEMOIRE RAM

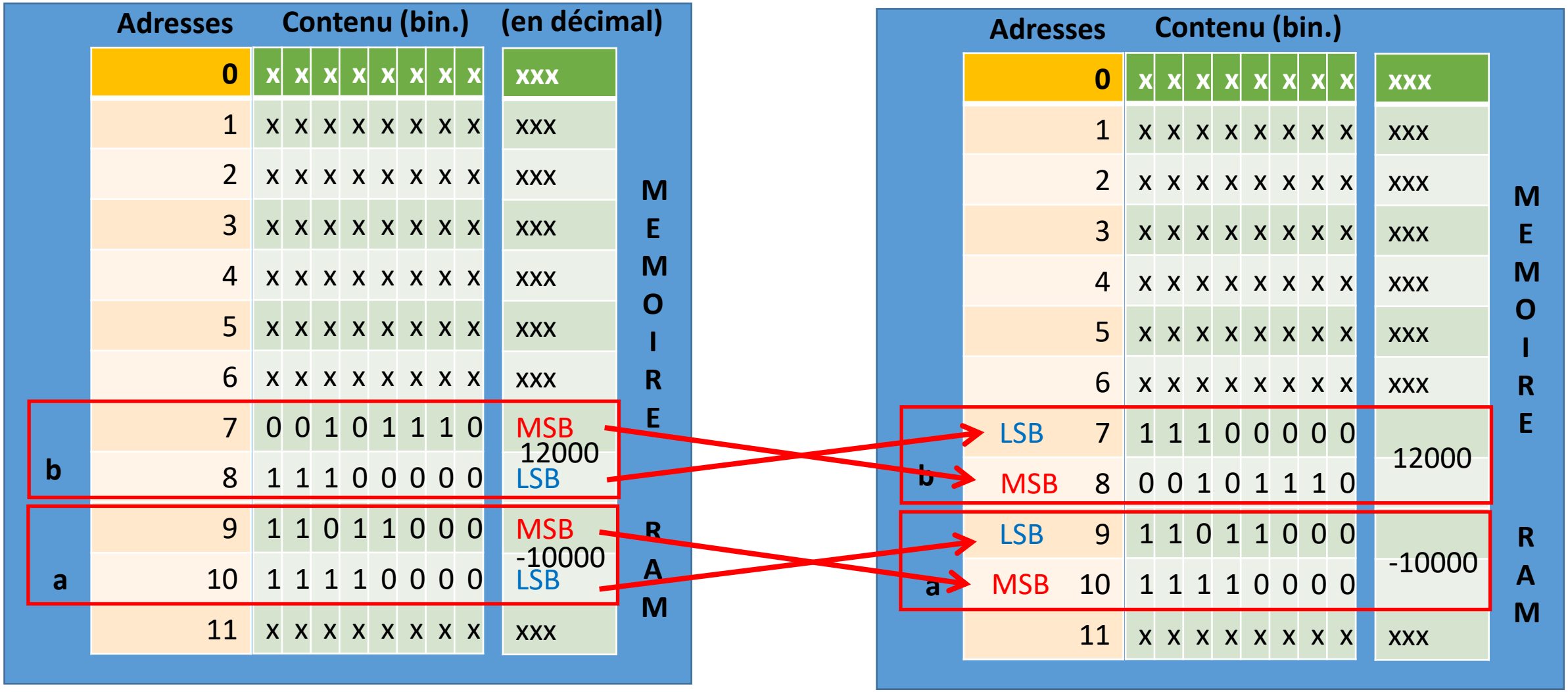
1 ligne = 1 octet, x : quelconque

(http://www.exploringbinary.com/twos-complement-converter/)

Les types entiers : endianess ➔dépend du µP

LITTLE ENDIAN = M.S.Byte aux petites @

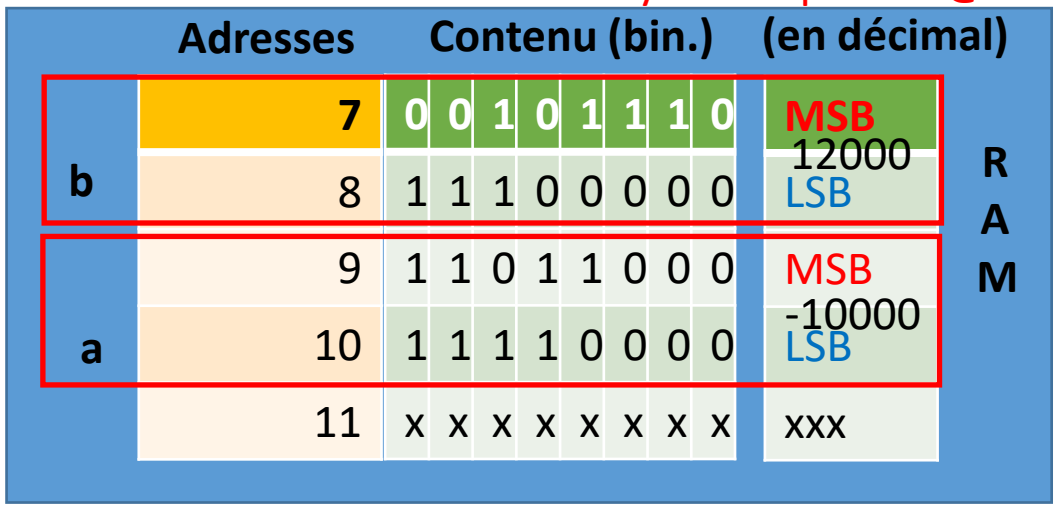
BIG ENDIAN = M.S.Byte aux grandes @



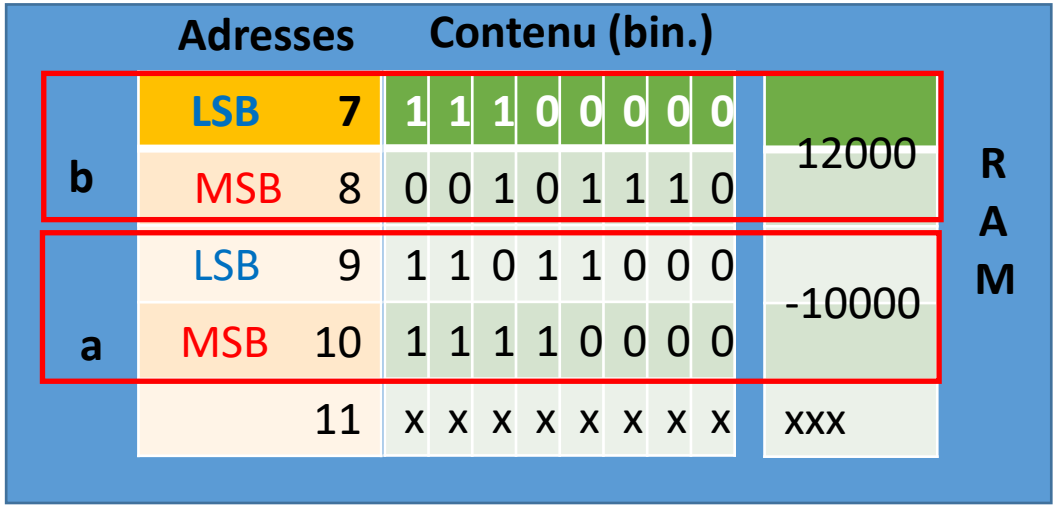
C'est totalement "transparent" pour le programmeur, on a pas à s'en soucier sauf dans certain cas (accès au matériel, réseaux)

Les types entiers : endianess , @

LITTLE ENDIAN = M.S.Byte aux petites @



BIG ENDIAN = M.S.Byte aux grandes @



printf("adresse de a=%p", &a);

➔adresse de a=9

printf("adresse de a=%p", &a);

➔ adresse de b=7

printf("adresse de a=%p", &a);

➔adresse de a=9

printf("adresse de a=%p", &a);

➔ adresse de b=7

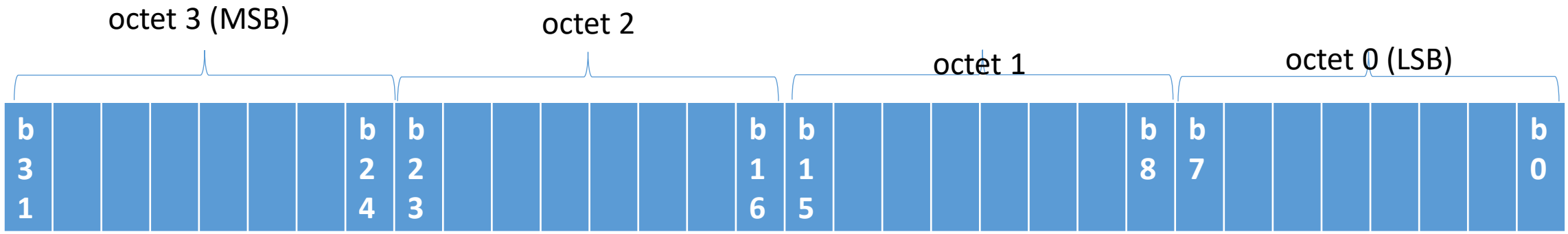
⟺

Identique

47

Les types entiers : 32bits (=4 octets)

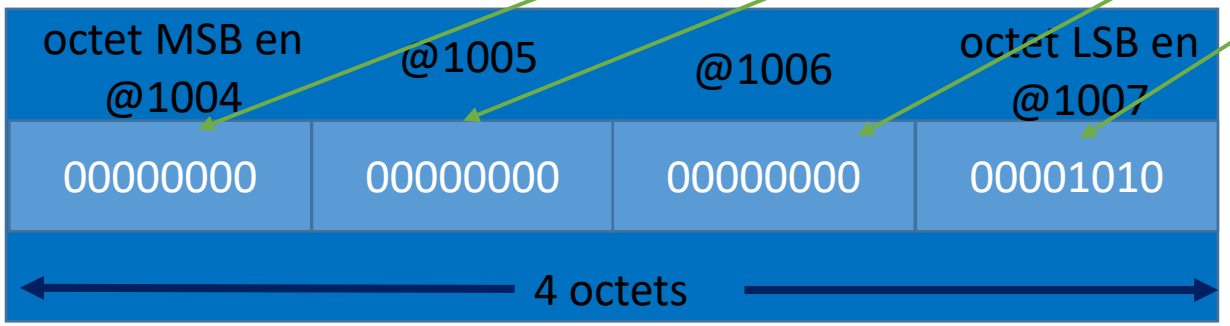
- **int**: 32 bits, codé en complément à 2, valeur entre ..... et .....
- **unsigned int,** valeur entre ..... et .....



- **Le type int fait souvent, mais pas toujours 32 bits, dépend du compilateur/µP :**
  - **peut faire 16, 24 ou 32 :**
    - µC : 16bits Texas Instrument MSP430 : sizeof(int) = 2 ⇔16 bits (http://mspgcc.sourceforge.net/manual/x987.html)
    - µP DSP 56000 : 24 bits
    - **sur PC : 32 bits**
  - **Pour savoir ce que l'on fait :**
    - **Utiliser sizeof(int)**

Les types entiers : int

```
#include <stdio.h>
int main () {
   int a = 10;
   unsigned int b= 12000;
   printf("a= %d b=%d", a,b);
   return 0;
}
```



	Adresses	Contenu (bin.) (en décimal)	
	997	x x x x x x x x	xxx
	998	x x x x x x x x	xxx
	999	x x x x x x x x	xxx
b	1000	0 0 0 0 0 0 0 0	(MSB)
	1001	0 0 0 0 0 0 0 0	12000
	1002	0 0 1 0 1 1 1 0	
	1003	1 1 1 0 0 0 0 0	(LSB)
a	1004	0 0 0 0 0 0 0 0	(MSB)
	1005	0 0 0 0 0 0 0 0	10
	1006	0 0 0 0 0 0 0 0	
	1007	0 0 0 0 1 0 1 0	(LSB)
	1008	x x x x x x x x	xxx
	1009	x x x x x x x x	xxx
	1010	x x x x x x x x	xxx
	1011	x x x x x x x x	xxx
	...		

4oct. (b)

4oct. (a)

MEMOIRE RAM

Le type entier : long

- A éviter car sa taille est variable
- pas forcément 8 octets
- pas normalisé, la taille dépend de l'OS, du processeur (32 bits/64 bits...)

Les types entiers : 64 bits (8 octets)

- **long long int** a;
- **unsigned long long int** b;
- Avec **C99** : **long long** et **unsigned long long**
- Attention en java : "long" = 64 bits (alors que C long = 32 bits)
- Attention affichage avec %L

Types entiers : usages dans l'embarqué

- Utiliser typedef pour définir des synonymes plus explicites :

  Usage : **typedef** NomTypeExistant NomNouveauType;

  - **Exemples :**
    - **typedef** char int8_t; → ensuite on peut écrire **int8_t b;**
    - **typedef** unsigned char uint8_t; → ensuite on peut écrire **uint8_t a;**
    - **typedef** short int16_t; → ensuite on peut écrire **int16_t d;**
    - **typedef** unsigned short uint16_t; → ensuite on peut écrire **uint16_t c;**
    - **typedef** unsigned int uint32_t; → ensuite on peut écrire **uint32_t e;**
    - **etc**

Types et structures de données du C

- Types entiers
  - char / unsigned char
  - affichage en hexa, opérateur sizeof, codage ASCII
  - short, endianness
  - int
- **Type flottants**
  - Limites du codage en virgule fixe
  - Codage IEEE 754 (float, double)
  - Avantage inconvénients : FPU…
- Tableaux
  - 1D, déclaration, initialisation, allocation mémoire, erreurs fréquentes
  - 1D Chaîne de caractères, bibliothèque string.h
  - 2D
- Structures
- Les pointeurs (1ère partie)

Types flottants : virgule fixe

- Comment coder les nombres à virgules ?
- Codage en virgule fixe :
  - on décide de la position de la virgule arbitrairement (en fonction de ses besoins)
  - chaque bit à droite de la virgule code l'inverse d'une puissance de 2 : 1/(2^n)

Ex :

0	0	1	0	0	0	1	1
bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
2^2 = 4	2^1=2	2^0=1	2^-1 =1/2^1 =1/2 = 0.5	2^-2 =1/2^2 =1/4 = 0.25	2^-3 =1/2^3 =1/8 = 0.125	2^-4 =1/2^4 =1/16 = 0.0625	2^-5 =1/2^5 =1/32 0.03125

Types flottants : virgule fixe vs virgule flottante

On peut ainsi coder :

- 3.5d=01110000b
- 3.75d=01111000b
- etc

Avantage : simple à utiliser, implanter car utilise les unités arithmétiques entière existantes

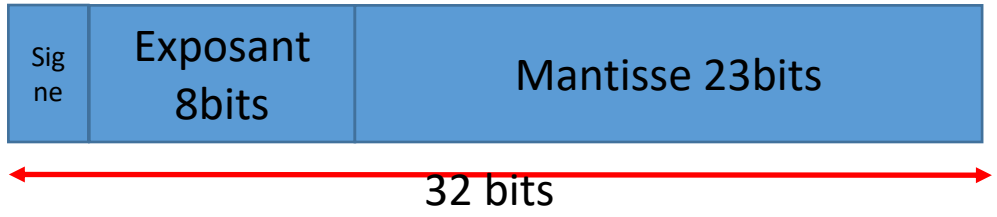
Inconvénient : précision et amplitude très très limitée

Types flottants : virgule flottante

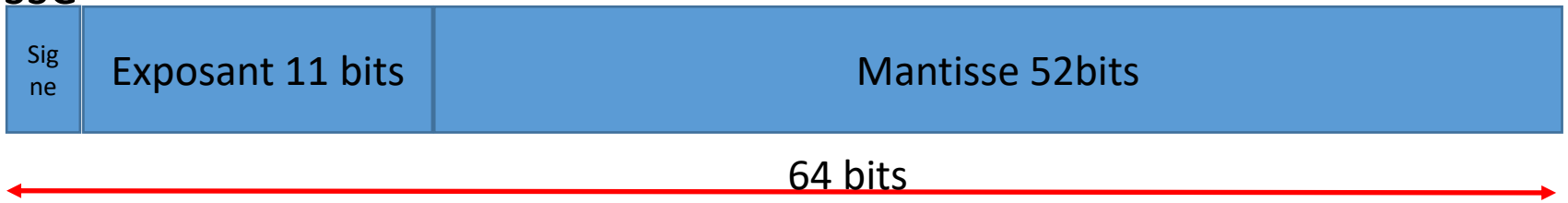
- Principe : compacter en codant chaque nombre par 2 parties
  - Mantisse
  - Exposant
- Exemple "intuitif" :
  - Ecrire 500000 ⇔ $0,5x10^6 = 5 \times 10^5$
  - Ecrire 65,1234 ⇔ 0.651234×10^2
  - Ecrire 0,000034 ⇔ $0.34 \times 10^{-3} = 3,4 \times 10^{-4}$
  - Ecrire -321 ⇔ $-0,321 \times 10^3$

Types flottants : virgule flottante **float, double**

- Norme IEEE 754, 2 types de flottants :
  - Simple précision : **float** ⇔ 32 bits : 1 bit signe, 8 bits exposant signé, 23 bit s mantisse
    - Codage exposant décalé de 127 (pour simplifier les comparaisons, pas de comp. à 2) va de -126 à +127
    - La mantisse code une valeur entre 1 et 2



  - Double précision : **double** ⇔ 64 bits : 1 bit signe, 11 bits exposant signé, 52 bits mantisse



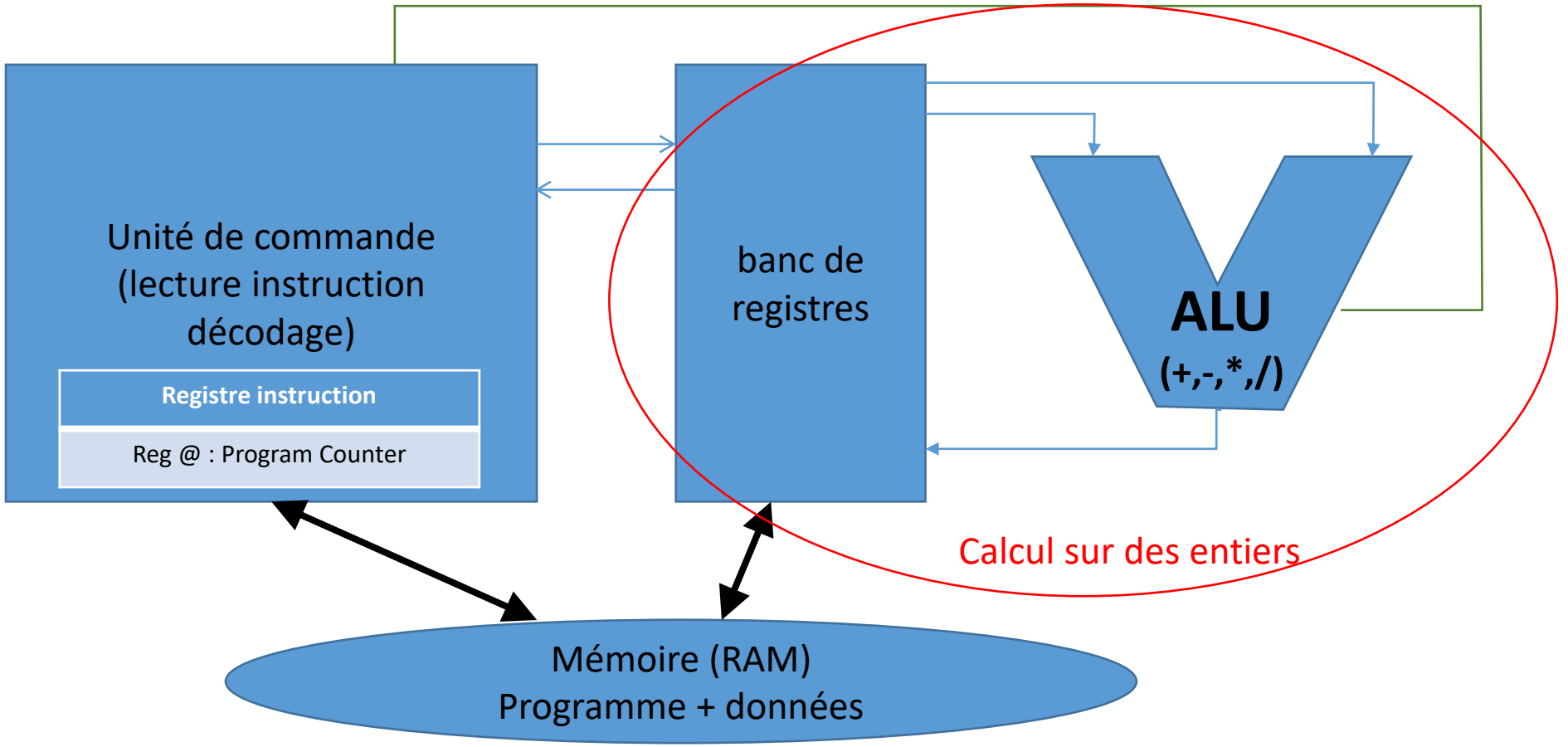
Types flottants : virgule flottante IEEE 754

- Il existe des valeurs particulières :
  - NaN : Not a Number , par exemple : "s111 1111 1xxx xxxx xxxx xxxx xxxx xxxx" avec X≠0 (résultat de 0/0, log(n<0)...)
  - + ∞ et -∞ : s111 1111 1 000 0000 0000 0000 0000 0000

  - Il existe aussi les long double : l'implémentation dépend du compilateur/OS/processeur

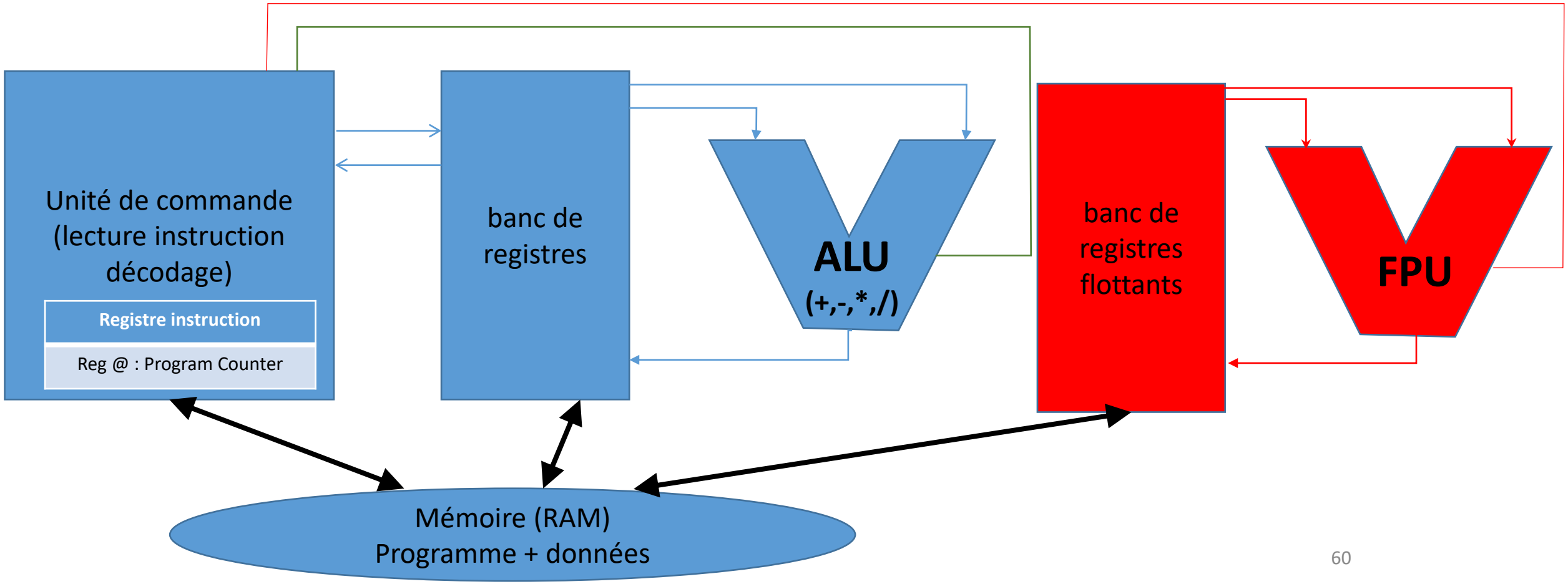
Types flottants : FPU

- Processeur sans FPU (la plupart des microcontrôleurs) :



Types flottants : FPU (Floating Point Unit)

- Processeur avec FPU : (MCU ARM Cortex M4)



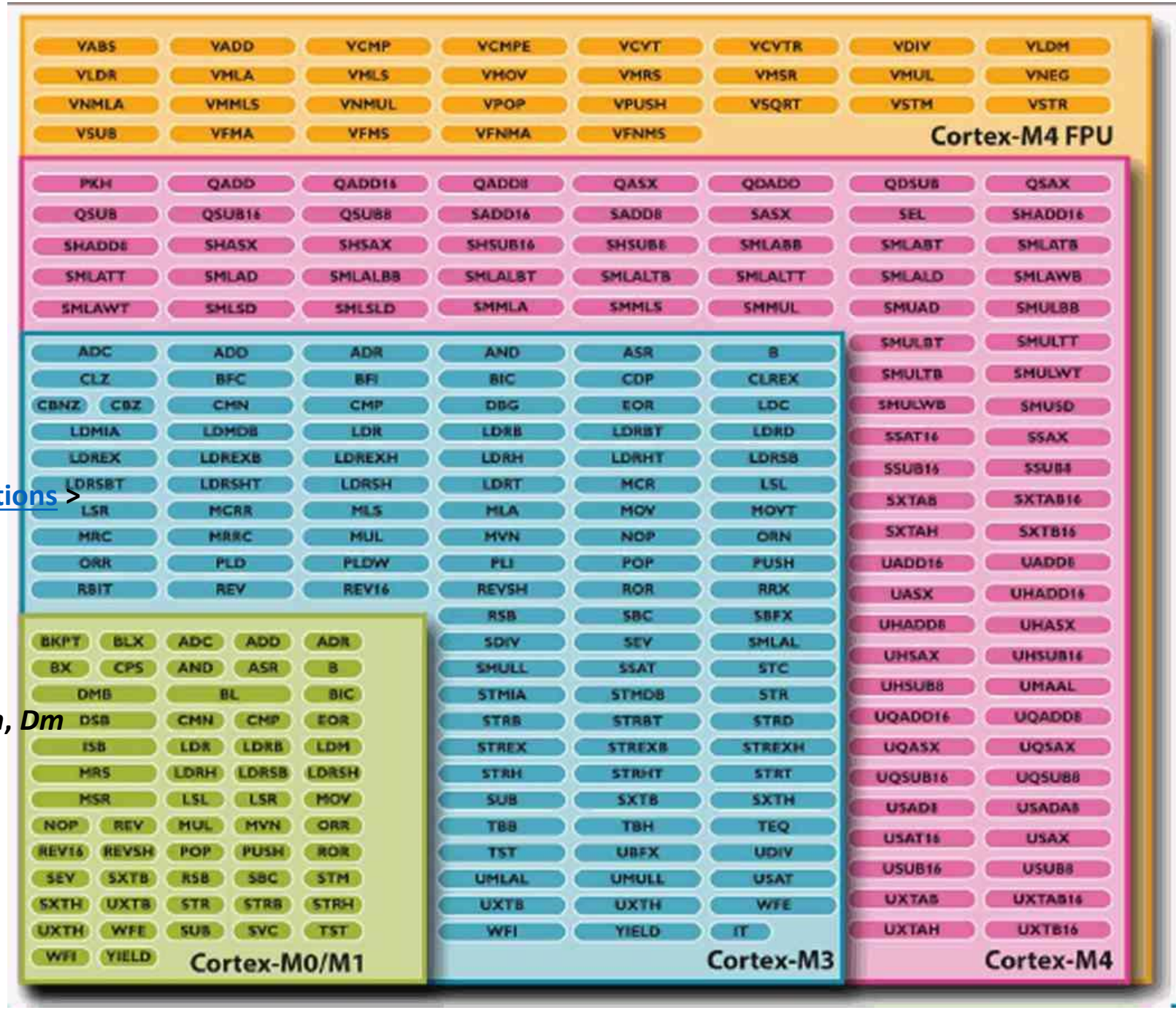
PROCESSEUR ARM

CORTEX M - Pour l'embarqué

M0 : + petit, faible consommation
M1, M3 : standard
M4 : avec possibilité de FPU (et inst. DSP)

ARM Compiler toolchain Assembler Reference Version 4.1
Home > NEON and VFP Programming > VFP instructions >

VADD, VSUB, and VDIV
Floating-point add, subtract, and divide.
These instructions can be scalar, vector, or mixed.
Syntax
V*op*{*cond*}.F32 {*Sd*}, *Sn*, *Sm* V*op*{*cond*}.F64 {*Dd*}, *Dn*, *Dm*



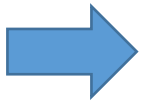
Types flottants : FPU vs ALU entière

- FPU : + de silicium, +coût, +gros, +consommation, +complexe
- ALU entière : + simple, -consommation, capable de traiter les flottants à l'aide de programme de bibliothèque, mais beaucoup plus lentement qu'une FPU !
- Exemple : famille microcontrôleur ARM Cortex M0, M1...M7

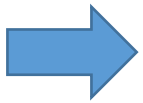
Attention conversion int⇔float et calcul

Qu'affiche l'extrait suivant ?

```
int w=15, x=2, y=0;
float z=0.0;

y=w/x;
printf("y=%d",y);
```

 .....

```
z=w/x;
printf("y=%f", z);
```

 .....

Types flottants : init., affichage

- Valeurs d'initialisation
  - double a=0.0, b=23.34, c = 45.3e-5, d = 0.123E-8;

- Pour float ou double : (préfixe l pour long double)
  - %f : affichage en notation décimale
  - %e : affichage en notation scientifique (exposant)
  - %g : affichage de la forme la plus compacte

  - Exemples :

float N = 12.1234;
printf("%f", N); //➔ 12.123400
printf("%e", N); // ➔ 1.212340e+01
double M = 12.123456789;
printf("%f", M); // ➔ 12.123457
printf("%e", M); ➔ 1.212346e+01
long double P = 15.5;
printf("%Le", P); ➔ 1.550000e+01

- Possibilité de choisir le nombre de digits à afficher

printf("%.2f", N);

Exemple :

```
#include <stdio.h>
int main(void) {
 float  x=3.123456789;
 printf("%.1f \n",x);        affiche 3.1
 printf("%.2f \n",x);        affiche 3.12
 printf("%.3f \n",x);        affiche 3.123
 return 0;
}
```

Comparaisons de float/double : DANGER

```
int main( ) {
  float a=0.1;
  float b=0.2;
  float res;


  res=a+b;


  printf("%f\n", a);
  printf("%f\n", b);
  printf("%f\n", res);

  if (res==0.3)
    printf("oui, a+b=0.3 \n");
  else
    printf("non, a+b est different de 0.3 \n");

  return 0;
}
```

(Cf. TP E1 Java avec epsilon)

Chapitre 2 : Types et structures de données

- Types entiers
  - char / unsigned char
  - affichage en hexa, opérateur sizeof, codage ASCII
  - short, endianness
  - int
  - long et long long int
- Type flottants
  - Limites du codage en virgule fixe
  - Codage IEEE 754 (float, double)
  - Avantages et inconvénients, problèmes de précision, temps d'execution : FPU…
- Tableaux
  - 1D, déclaration, initialisation, allocation mémoire, erreurs fréquentes
  - 1D Chaîne de caractères, bibliothèque string.h
  - 2D
- Structures et unions

Les tableaux : une dimension (1D)

- Déclaration : **type NomTab[NbElt];** N'initialise pas le contenu du tableau

  int notes[100];
  float temperatures[50];

- Modification/lecture :

  notes[4]=10;
  t= note[4]; (avec int t; préalablement)

- **Les Indices partent de 0 donc indice MAX = Nb elements-1**
- Initialisation possible uniquement lors de la déclaration (mais pas obligatoire, dans ce cas le contenu non définit)

  char tab1[5]={10,20,30,40,50};
  short tab2[ ]={0,0,0}; ➔ **la taille est déduite de la liste d'initialisation donc ici ⇔ char tab2[3]={0,0,0};**
  int tab3[100]={1}; ➔ **tab3[0] vaut 1, et tous les autres entiers sont initialisés à 0**

- La taille en octet occupée par un tableau en mémoire = nombre d'elt * taille_d'un_elt
- L'opérateur sizeof retourne cette taille :

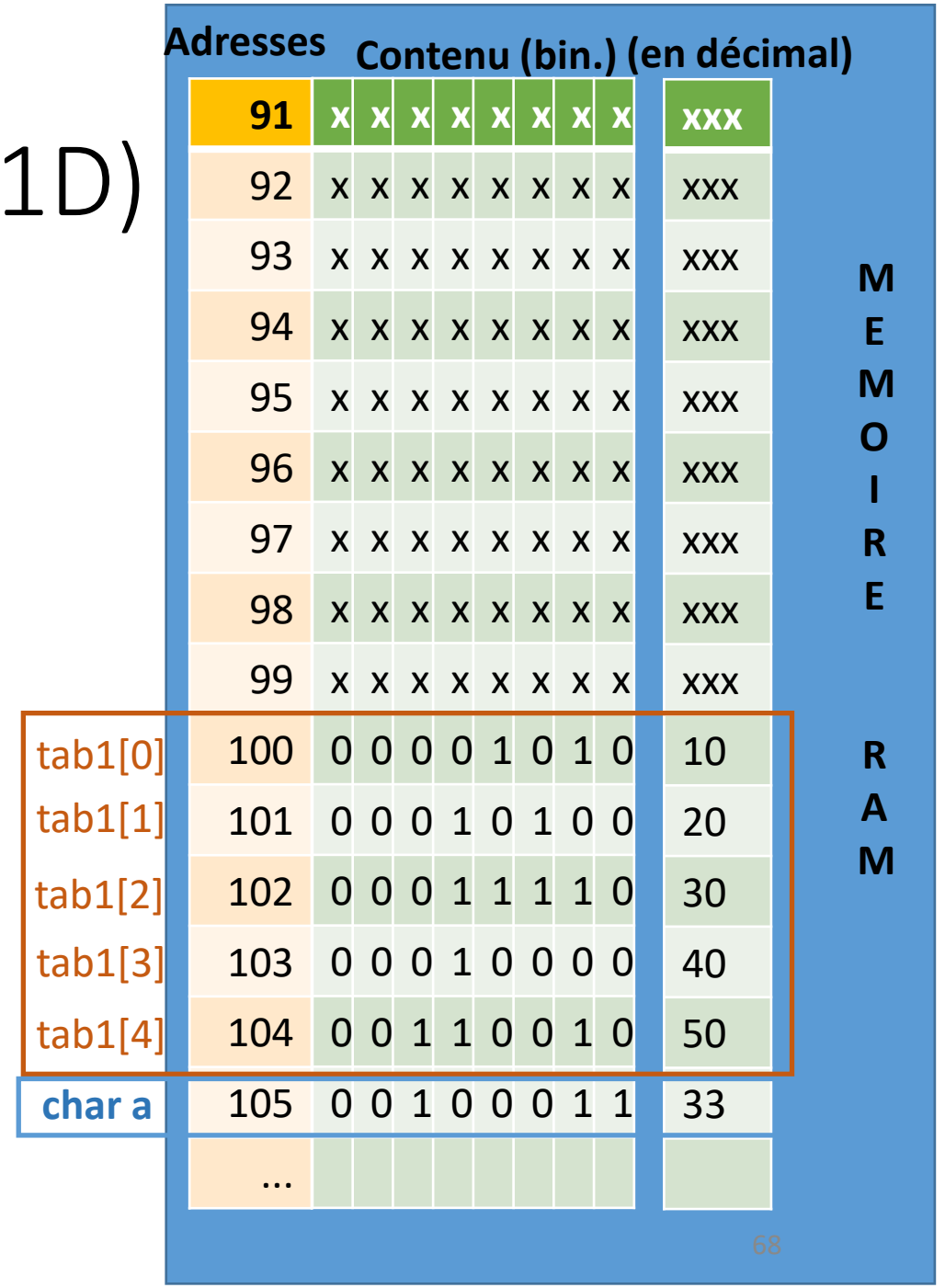
  sizeof(tab1) ?
  sizeof(tab2) ?
  sizeof(tab3) ?

Les tableaux : une dimension (1D)

```
char a= 33;
char tab1[5]={10,20,30,40,50};
```

Définition :

indices croissants⇔adresses croissantes



Les tableaux : une dimension (1D) adresses mémoire

int tab[3]={10,20,30};

printf("adresse tab[0] : %p", &tab1[0]); ➔1000
printf("adresse tab[1] : %p", &tab1[1]); ➔1004
printf("adresse tab[2] : %p", &tab1[2]); ➔1008

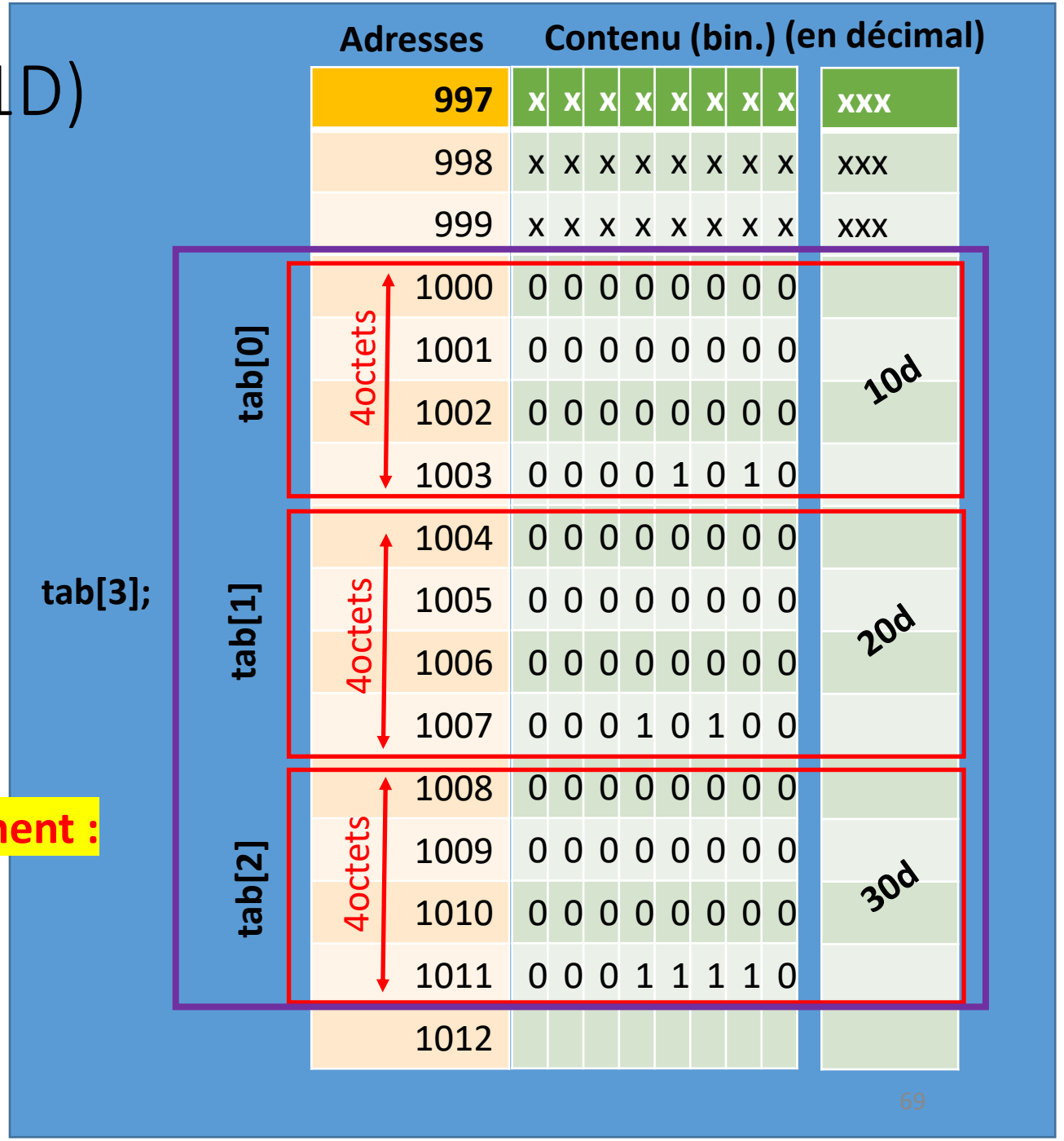
Définition :

le nom d'un tableau est l'adresse de son 1er élément :

tab⇔ &tab[0]

donc ici :

printf("adresse tab : %p", tab1); ➔ 1000



Les tableaux : une dimension (1D)
Les dangers (prudence)

- On ne peut pas afficher tous les éléments du tableau avec un seul printf : faire une boucle
- La seule façon d'initialiser plusieurs éléments du tableau est au moment de sa déclaration, ensuite on ne peut accéder (en lecture ou écriture) que élément par élément :
  - *int tab[3]={10,20,30};* //valide
    - alors que :

      int tab[3];

      tab[3]={10,20,30}; **est non valide**
- On ne peut copier un tableau avec le signe =, même au moment de sa déclaration
  - On ne peut pas écrire :

    int tab1[3]={10,20,30};

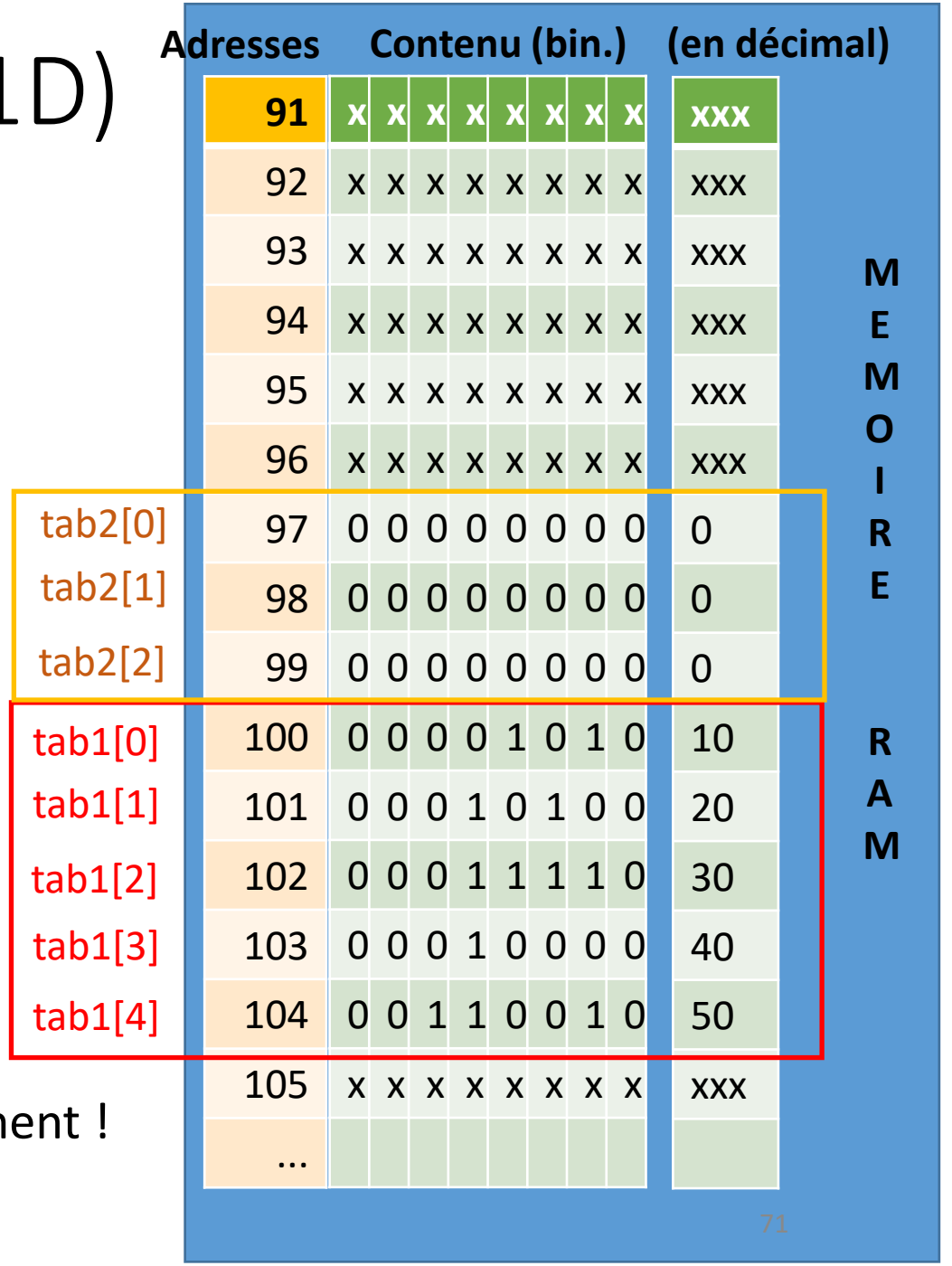
    int tab2[3] = tab1; //non valide (on verra pourquoi dans le chapitre pointeurs)
- Il n'y a aucun moyen de connaitre la taille d'un tableau en dehors de la fonction qui l'a créé (si on passe un tableau à une fonction, la fonction n'a aucun moyen de connaitre la taille de se tableau, le sizeof retourne autre chose…) on verra pourquoi dans le chapitre pointeurs

Les tableaux : une dimension (1D) cas du dépassement de taille

```
#include<stdio.h>
int main() {
    char tab1[5]={10,20,30,40,50};
    char tab2[3]={0}; /*indices valides :0,1 et 2 */

    printf("tab1[0]=%d", tab1[0]); ➔ tab1[0]=10

    tab2[3] = 0; /*dépassement indice max..*/

    printf("tab1[0]=%d", tab1[0]); ➔ tab1[0]=255
```

erreur

Attention, le compilateur **n'a pas vu d'erreur**, pas d'avertissement....et tab1 à été modifié accidentellement !



Tableaux 1D de caractères = chaînes de carac.

(Rappel : char c='A'; ⇔ char c=65; et affichage avec %d ➔ 65 et %c➔ A)

- On veut afficher la chaine de caractère ABC

  1ère idée : char tab[]={65,66,67};
  ⇔ char tab[ ]={'A','B', 'C'};

- Pour afficher le contenu de tab, faire une boucle et afficher 1 à 1 les caractères :

```
int i;
char tab[] = {'A','B','C'};
for (i=0;i<sizeof(tab)-1; i++)
      printf("%c", tab[i]);
```

Inconvénient : il faut connaitre la taille du tableau, le nombre d'éléments du tableau

Tableaux 1D de caractères = chaînes de carac.

Pb : on a vu que utiliser sizeof pour la taille est dangereux **dans une fonction sizeof car sizeof** retourne autre chose dans une fonction (Cf. pointeurs)

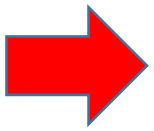
Solution adoptée en C : un caractère supplémentaire ajouté en dernier pour indiquer la fin de la chaîne : ce sera '\0' (=caractère NULL de code ASCII 0)

➔ il faut donc une taille plus grande de 1 élément quand on alloue le tableau

ex. : char tab[ ] = {'A','B','C', '\0' }; ➔ taille = 4 octets

Permet ensuite d'afficher tous les caractères sans connaitre la taille à priori :

```
int i=0;
while (tab[i] != '\0'  ) {
        printf("%c", tab[i]);
        i++;
}
```



Problème : c'est fastidieux accolades, + guillemets + faire une boucle pour l'affichage...

Tableaux 1D de caractères : déclaration chaînes de caractères

Pour déclarer une chaîne de caractères plus simplement : utilisation des guillemets

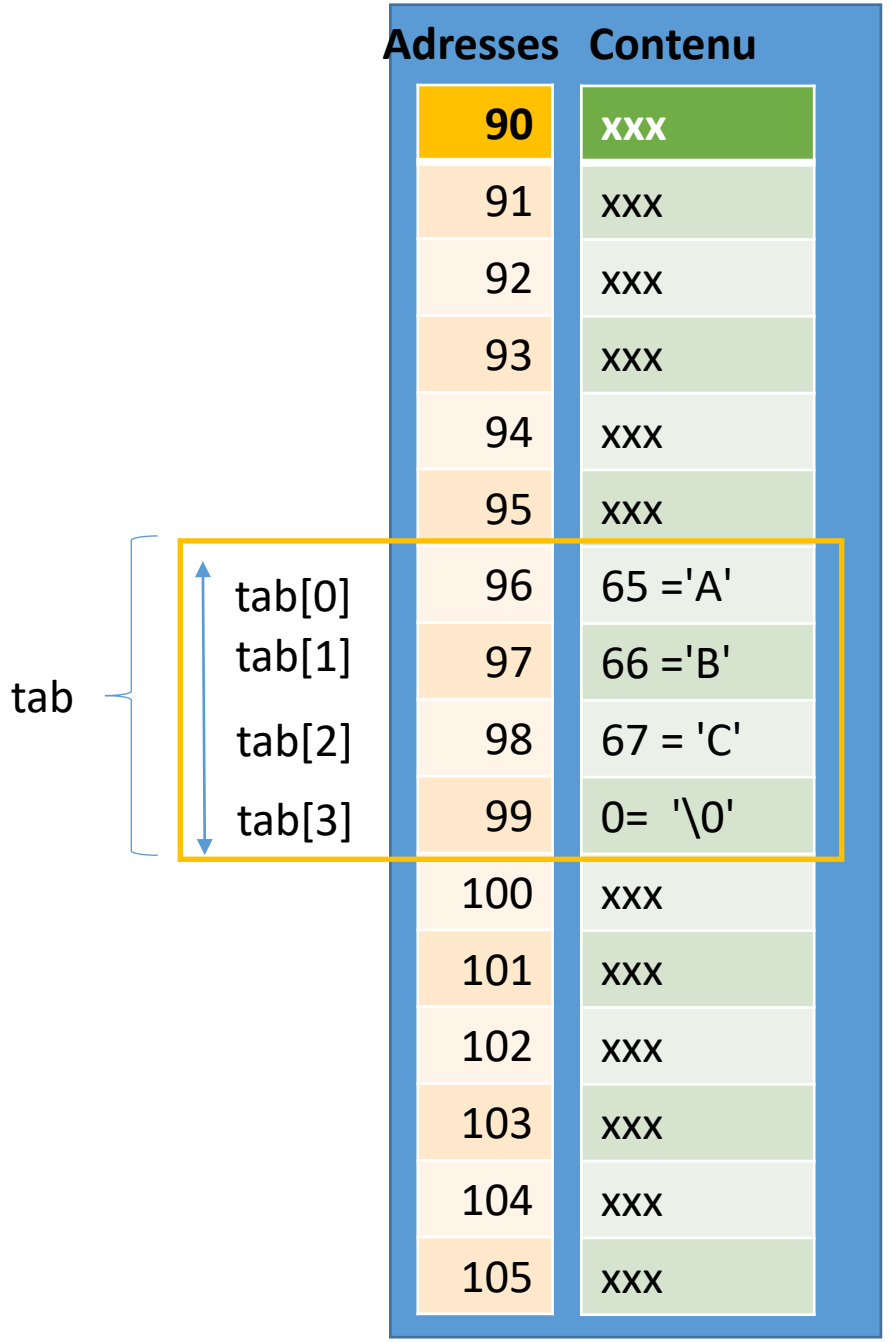
char tab[]="ABC"; ➔ le caractère \0 est automatiquement ajouté

donc sizeof(tab) = 4 octets!

Donc les guillemets :

1. **permettent d'éviter d'écrire un à un les caract. entre apostrophes**
2. **permettent d'éviter d'écrire les accolades**
3. **ajoutent automatiquement \0 à la fin**

et pour l'affichage...?



Tableaux 1D de caractères : affichage chaînes de caractères

Pour afficher toute la chaine en une fois : %s

```
char tab[]="ABC";   (⇔ char tab[ ]={'A','B','C','\0'}; )
printf("la chaîne est : %s",tab);
```

➔ affiche 1 à 1 les caractères jusqu'à \0

bien sûr on peut toujours le faire avec une boucle et *printf ("%c",tab[i])*

Entre apostrophes simples : ' ' ⇔ pour 1 seul caractère
Entre guillemets : " " pour 1 ou plusieurs caractères **suivi de \0 ajouté automatiquement**

Tableaux 1D de caractères : Confusion entier/chaine

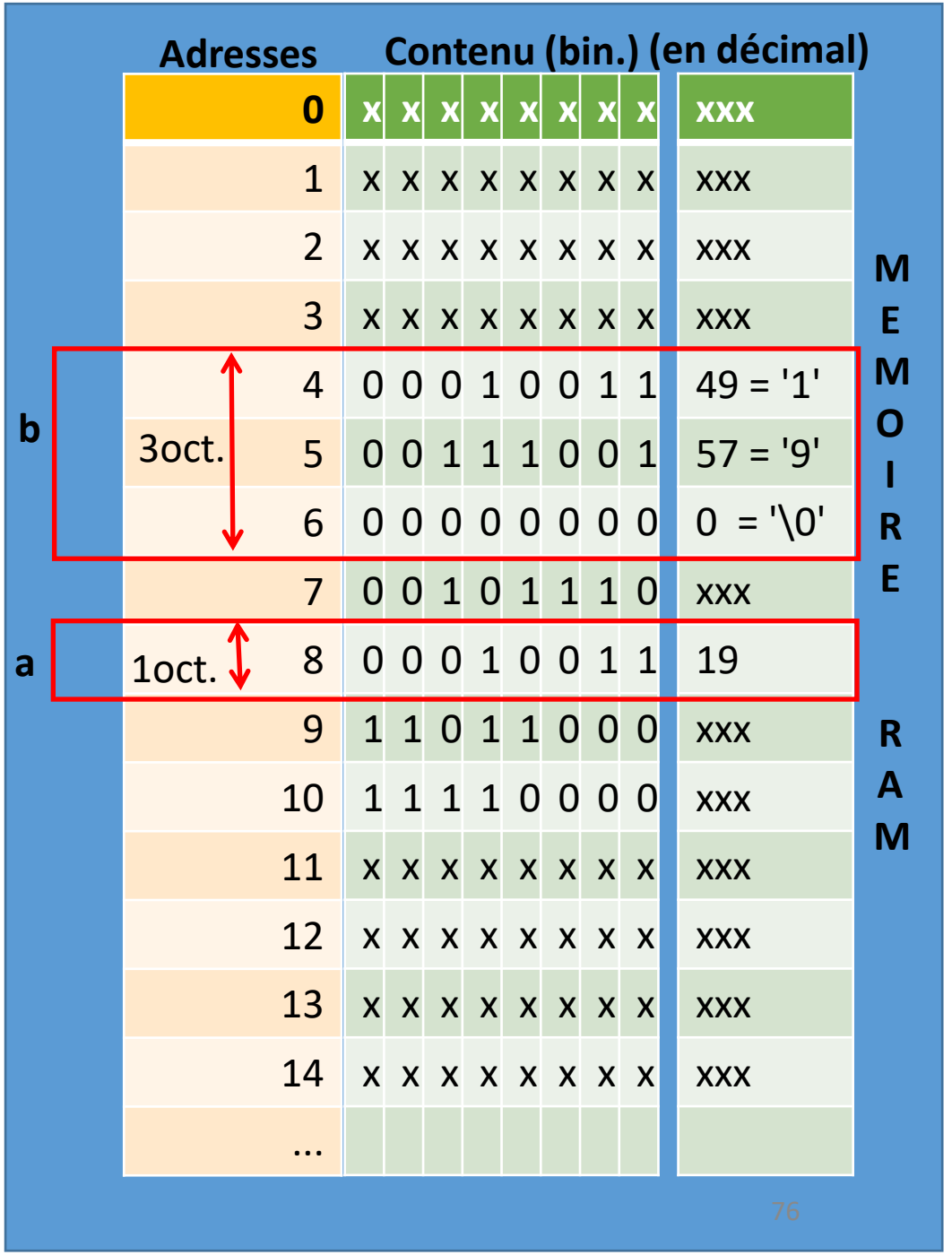
char a = 19;

Ne pas confondre avec :

char b[] = "19";

⇔ char b[]={'1','9','\0'};

⇔char b[]={49, 57,'\0'};



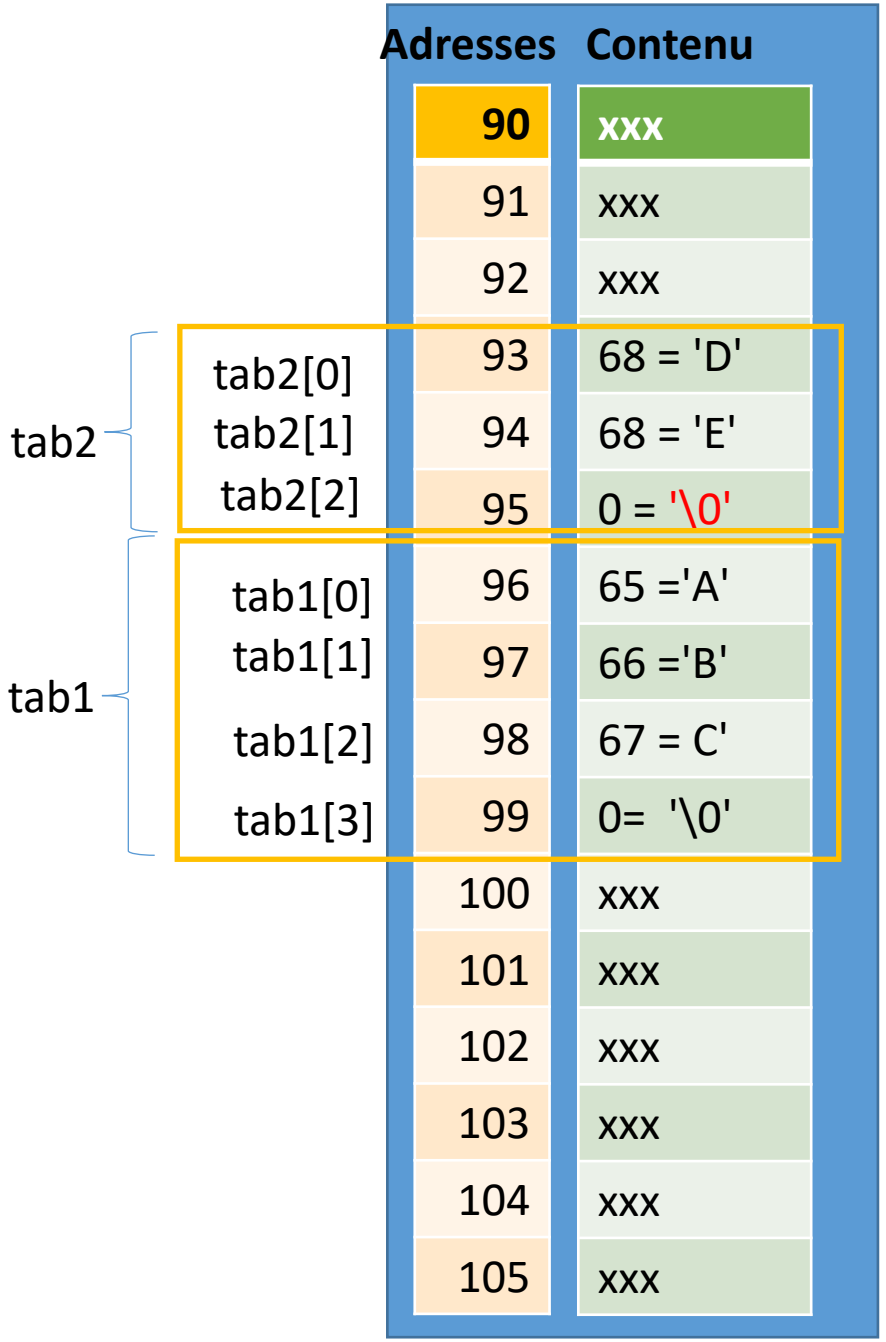
Tableaux de caractères :

```
char tab1[] = "ABC";
char tab2[3] = "DE";
printf("%s", tab1); ➔ affiche ABC
printf("%s", tab2); ➔ affiche DE
```

Que ce passe t-il si on modifie tab2[2] :

```
char tab1[] = "ABC";
char tab2[3] = "DE";
tab2[2]='Z'; //on écrase le caractère \0
printf("%s", tab1); ➔ affiche ABC
printf("%s", tab2); ➔ affiche :
```

Et si on écrase tab1[3] ?



Tableaux de caractères : taille

Var1 = strlen (Nom_tableau); ➔ retourne dans Var1 le nb de carac. i.e : le nombre de caractères avant le caractère null '\0' (**STR**ing **LEN**gth)

```
#include<stdio.h>
#include <string.h>
int main() {
        char tab1[30] = "ABC";              // quelle est le nombre de caractères dans tab1 ?
        int taille;
        taille = strlen (tab1);
        printf("%s \n", tab1);                                  // affiche ……..?
        printf("nb caract. : %d \n", taille);                   // affiche ……..?
        printf("taille en octet : %d \n", sizeof(tab1));        // affiche ……..?

        strcpy(tab1, "ESIEE");
        taille = strlen (tab1);
        printf("%d \n", taille);                // affiche ……..?
        printf("chaine : %s \n", tab1);         // affiche ……..?
        return 0;
}
```

Rem. : si l'on veut modifier tab1 : ne pas mettre des crochets vides, car alors ⇔ char *tab1="ABC"; alloué ailleurs..

Tableaux de caractères : recopie

char tab1[6]="Hello";
char tab2 [6];
En C on ne peut pas écrire : tab2 = tab1;
en effet on a vu que

tab2 ⇔ &tab2[0] *par exemple printf("%p",tab2); affiche 93*
tab1 ⇔ &tab1[0] *par exemple printf("%p",tab1); affiche 96*

donc écrire tab2=tab1; ⇔ &tab2[0] = &tab1[0]; ⇔ 93 = 96 !! ??

➔ AUCUN SENS !! (on ne peut changer l'@ du tableau)

De même si on déclare : char tab3[10];

écrire tab3=tab1+tab2; n'a pas de sens non plus !

Tableaux de caractères : recopie

Pour copier tab2 dans tab1 il faut donc copier élément par élément jusqu'a atteindre le caractère fin de la chaîne (en vérifiant en même temps que la taille du tableau destinataire est assez grande).

1) Solution avec une boucle :

```
i=0;
tant que ( (tab1[i] != '\0') && (i < sizeof(tab2)-1) ) {      (rappel : && ⇔ ET logique)
        tab2[i] = tab1[i];
        i++;  /* ⇔ i=i+1; */
 }
```

2) Solution 2 : utiliser la fonction **strcpy** de la bibliothèque **string.h** :

strcpy(nom_tab_dest, nom_tab_source); /* **STR**ing **C**o**PY** */

```
#include<string.h>
...
char tab1[]="Hello";
char tab2[50];
strcpy(tab2,tab1);
```

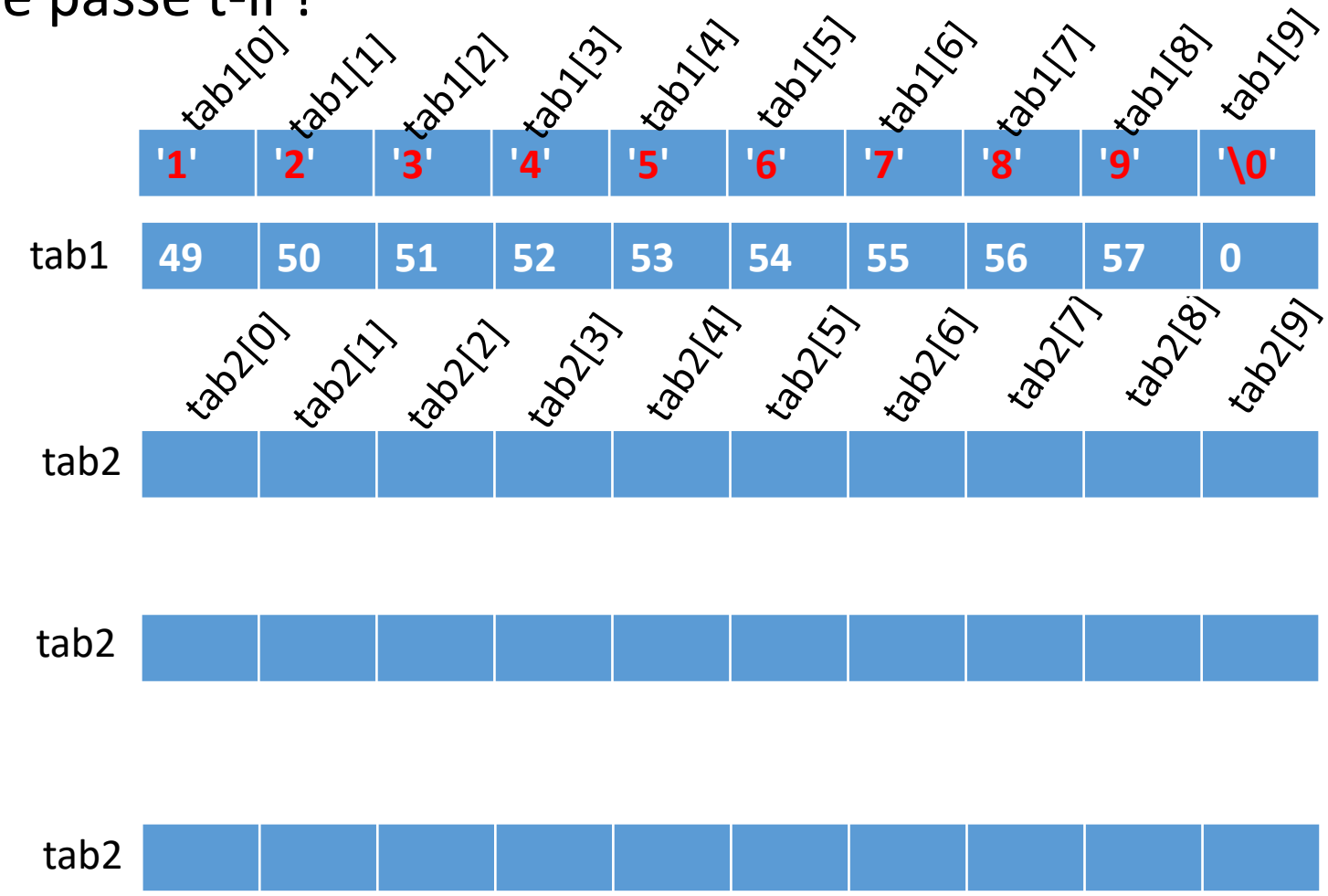
➔ Attention : rappel la fonction stcpy ne peut connaitre le nombre d'octets réservés pour le tableau destinataire (le sizeof ne retourne pas la taille d'un tableau quand il est passé en argument) ➔ risque de bug si on a pas pris soin de vérifier avant.

(pour éviter cela il existe une variante plus sécurisée : **strncpy** mais n'ajoute pas '\0' à la fin, faire "man strncpy" pour avoir la page de manuel)

Tableaux de caractères : recopie

Exemples : 2 strcpy successifs, que se passe t-il ?

```
char tab1[10]="123456789";
char tab2[10];

strcpy(tab2,tab1);
printf("%s",tab2);

strcpy (tab2, "ESIEE");
printf("%s",tab2);

tab2[5]='Z';
printf(" %s",tab2);
```



Tableaux de caractères : comparaison

De même qu' **on ne peut pas écrire** tab2 = tab1;

on ne peut pas comparer 2 chaines en écrivant :

if (tab1 == tab2) ….. //cela test quoi ?

//➔ que valent tab1 et tab2 ?

Pour comparer 2 chaînes :

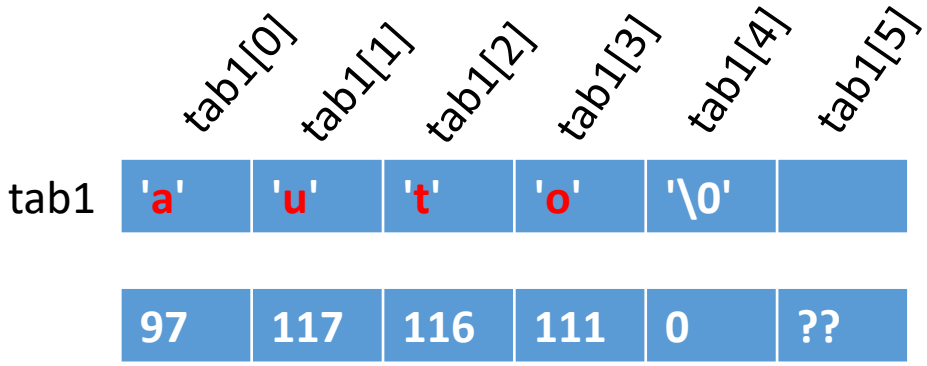
il faut comparer 1 à 1 successivement les élèments du tableau avec une boucle

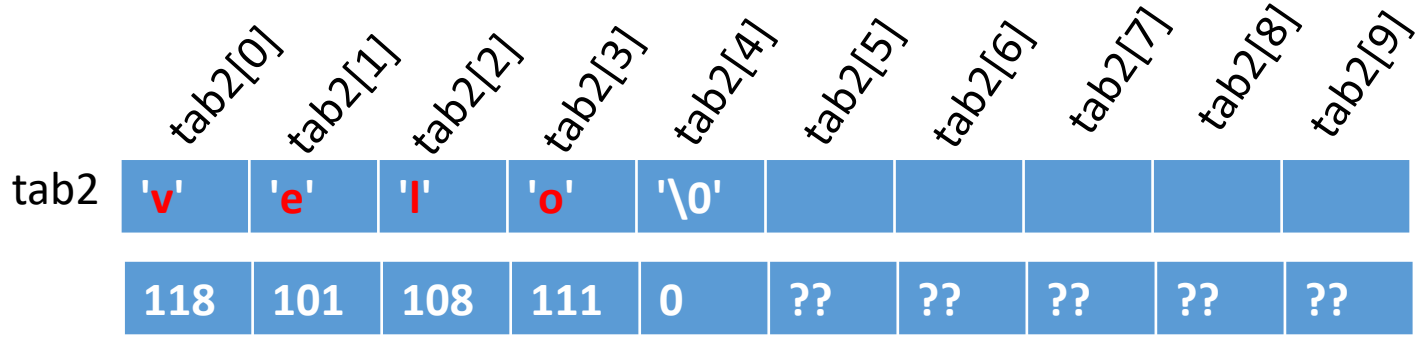
ou utiliser la fonction STRing CoMPare (chaine 1, chaine 2)

Attention : différence avec JAVA

Tableaux de caractères : comparaison

```
#include<stdio.h>


int main () {
        char tab1[6]="auto", tab2[10]="velo";

        int i=0, difference=0;

        while ( (tab1[i]!='\0') || (tab2[i]!='\0') ) {
                if (tab1[i] != tab2[i])
                                difference=1;
                i++;
        }
        if (tab1[i] != tab2[i] ) //il faut la même taille
                difference = 1;

        if (difference == 1)
                printf("Les chaines sont différentes");
        else
                printf("Les chaines sont identiques");
        return 0;
}
```





Tableaux de caractères : comparaison

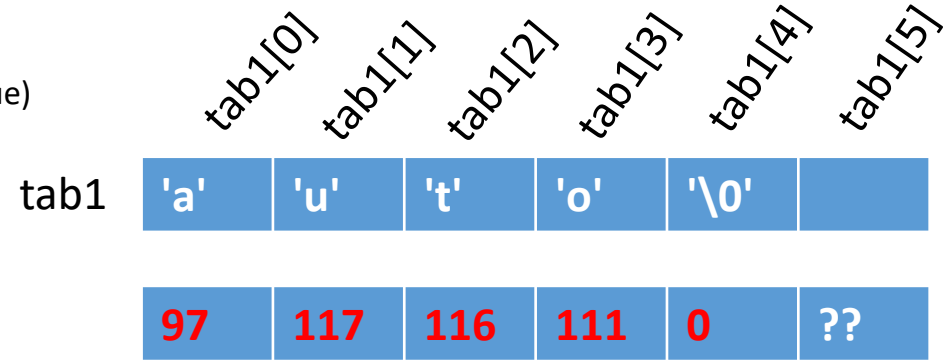
fonction STRing CoMPare (chaine 1, chaine 2)

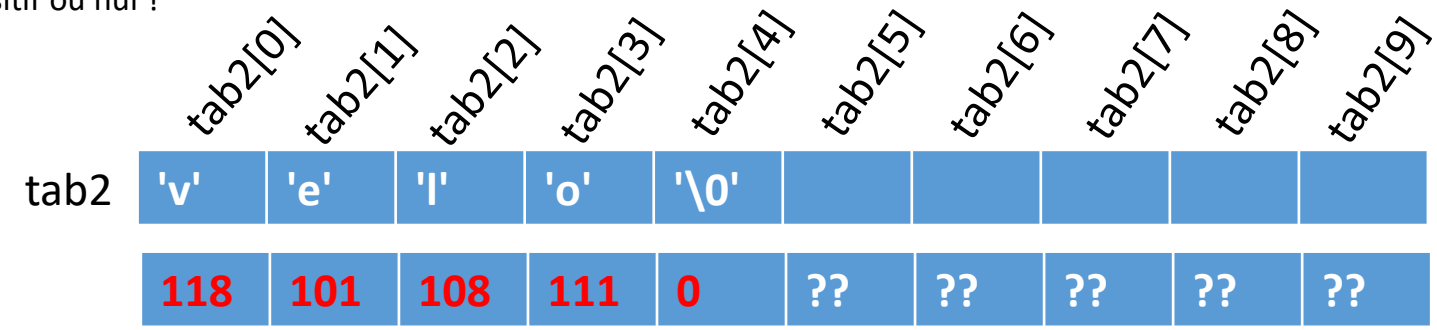
permet de classer dans l'ordre alphabétique

strcmp retourne un entier :

- 0 : si les deux chaines sont identiques
- <0 : si chaine 1 < chaine 2 (dans l'ordre lexical/alphabétique)
- >0 : si chaine 1 > chaine2

```
#include<stdio.h>
#include <string.h>
    int main () {
    char tab1[6]="auto", tab2[10]="velo";
    int a;
    a = strcmp(tab1, tab2);
    printf ("a=%d", a);              //Est-ce que a est négatif, positif ou nul ?
    return 0;
}
```





Tableaux : piège, erreurs classiques

Seules les chaines de caractères se terminent pas un '\0',

➔ les autres tableaux (int, float etc) n'ont pas d'élément supplémentaire pour marquer la fin

float tab[]= {10.0, 20.0, 30.0, 40.0}; donc pas de caractère '\0' ajouté à la fin

NE PAS ECRIRE :

- int tab[]; //initialisation sans taille…compile..mais ne réserve pas de mémoire (Cf. pointeurs)
- tab1=tab2; //FAUX, car nom de tab = @ de son 1er elt
- if (tab1==tab2) est FAUX de même if (tab1=="toto") est FAUX

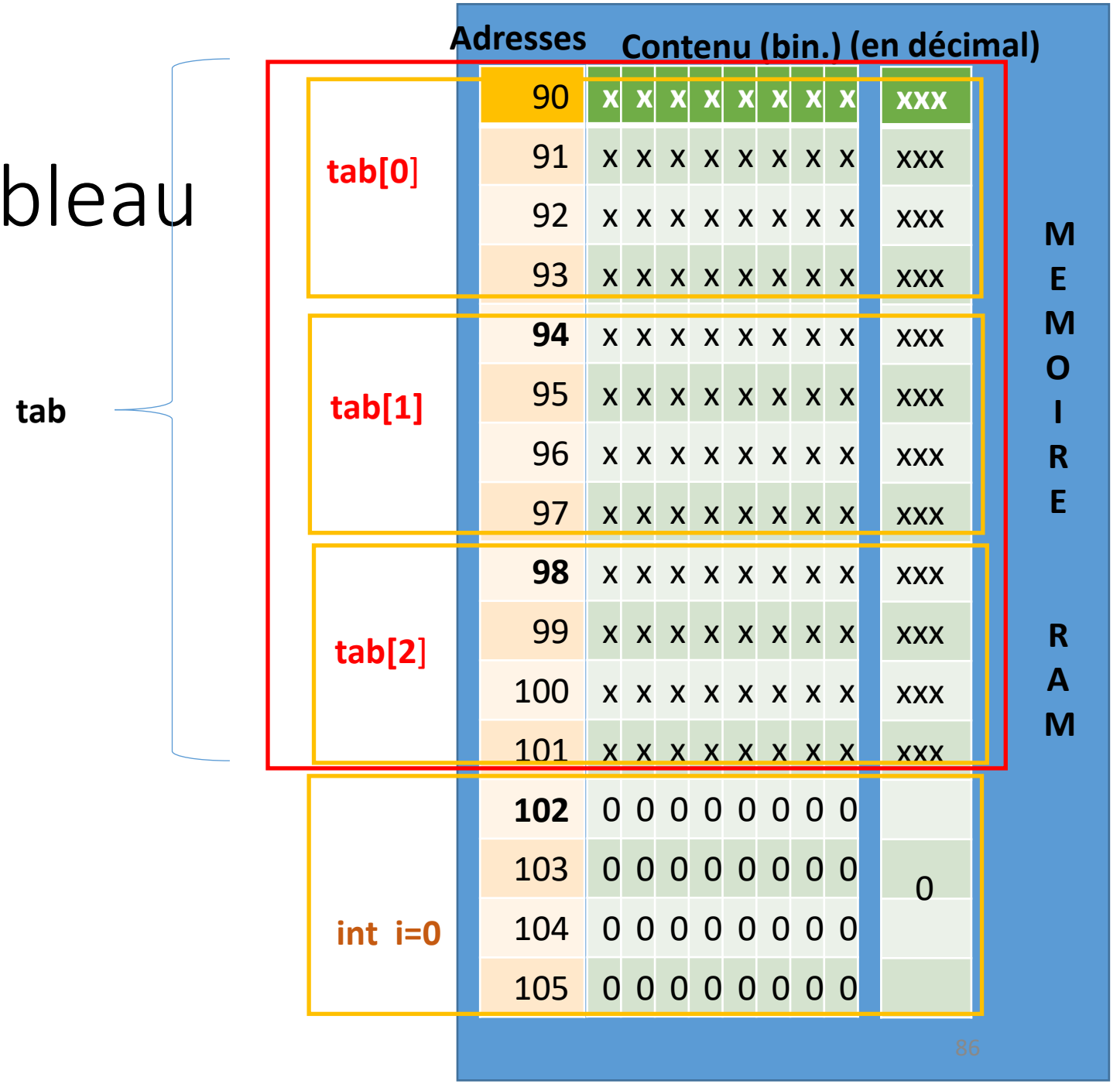
Les tableaux 1D : dépassement de tableau

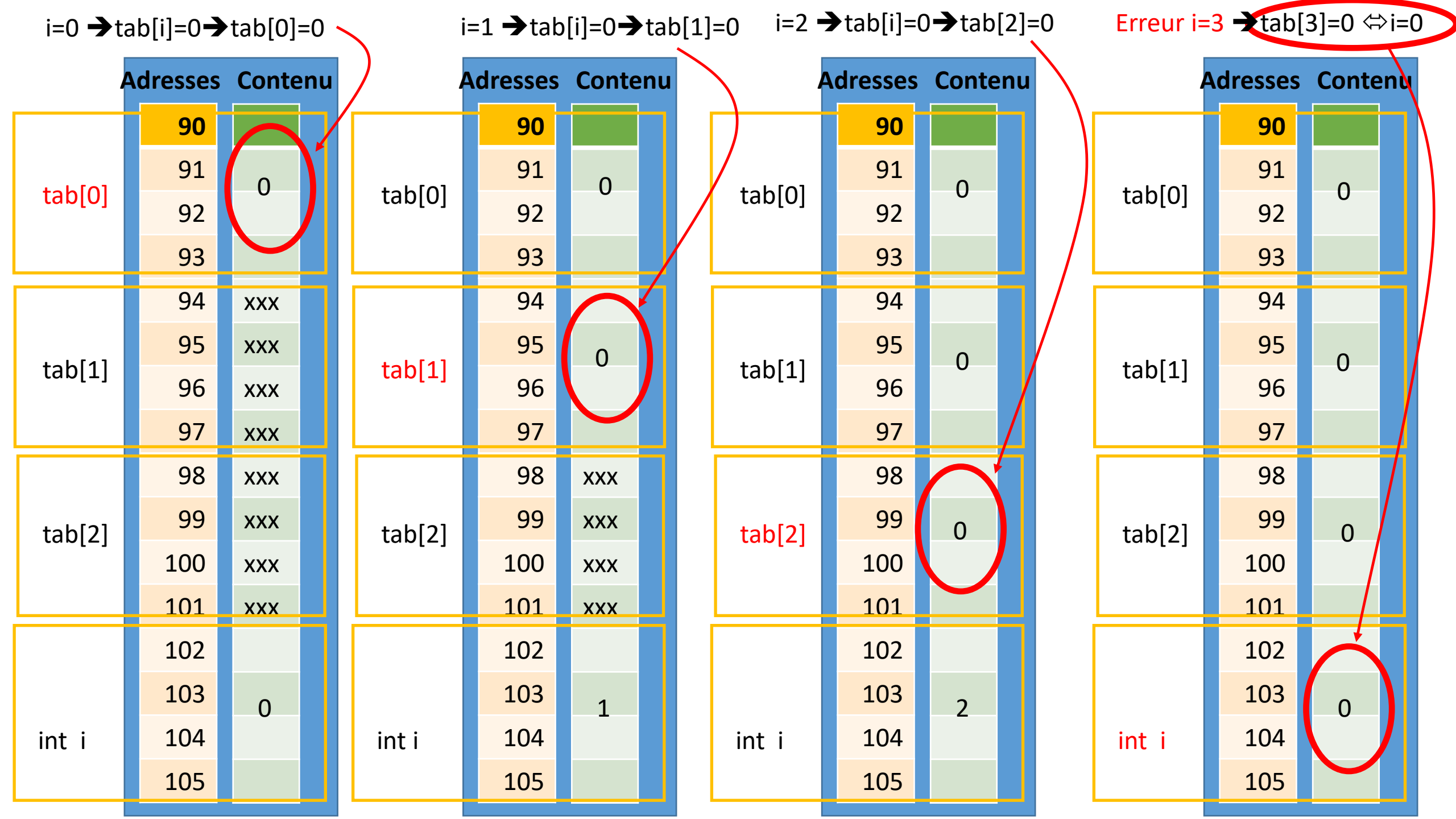
```
#include<stdio.h>
int main() {
      int i;
      int tab[3];
      for (i=0;i<=3;i++)
          tab[i]=0;
      return 0;
}
```

Que va t-il se passer ???

N.B: sizeof(int) = 4 octets

donc sizeof(tab) = ??



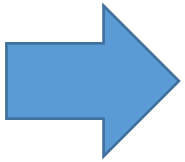

i=0 ➔tab[i]=0➔tab[0]=0
i=1 ➔tab[i]=0➔tab[1]=0
i=2 ➔tab[i]=0➔tab[2]=0
Erreur i=3 ➔tab[3]=0 ⇔i=0
Adresses Contenu
tab[0]
tab[1]
tab[2]
int i
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
0
xxx
1
2

Tableaux 1D : dépassement d'indice

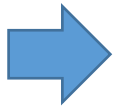
- Buffer Overflow
- Le compilateur ne les détecte pas (ils arrivent à l'exécution), pas d'erreur, pas de warning
- Si l'adresse écrasée appartient au programme : risque de bug (boucle infinie, comportement erratique, comportement correct sauf pour certaine valeur...)
- Si l'adresse écrasée n'appartient pas au programme : interception de l'accès mémoire par l'Operating System (grâce à l'unité "MMU")
  - Sous Windows : fenêtre "Access violation"
  - Sous Linux : Segmentation fault
- Si pas d'OS (i.e. microcontrôleurs) : résultat imprévisible...
- Analyse de programme possible avec l'outils VALGRIND

Les tableaux : #define

```
#include<stdio.h>


int main() {
    int i;
    int tab[3];
    for (i=0;i<3;i++)
       tab[i]=0;
    return 0;
}
```



```
#include<stdio.h>
#define MAX 3

int main() {
    int i;
    int tab[MAX];
    for (i=0;i<MAX;i++)
       tab[i]=0;
    return 0;
}
```

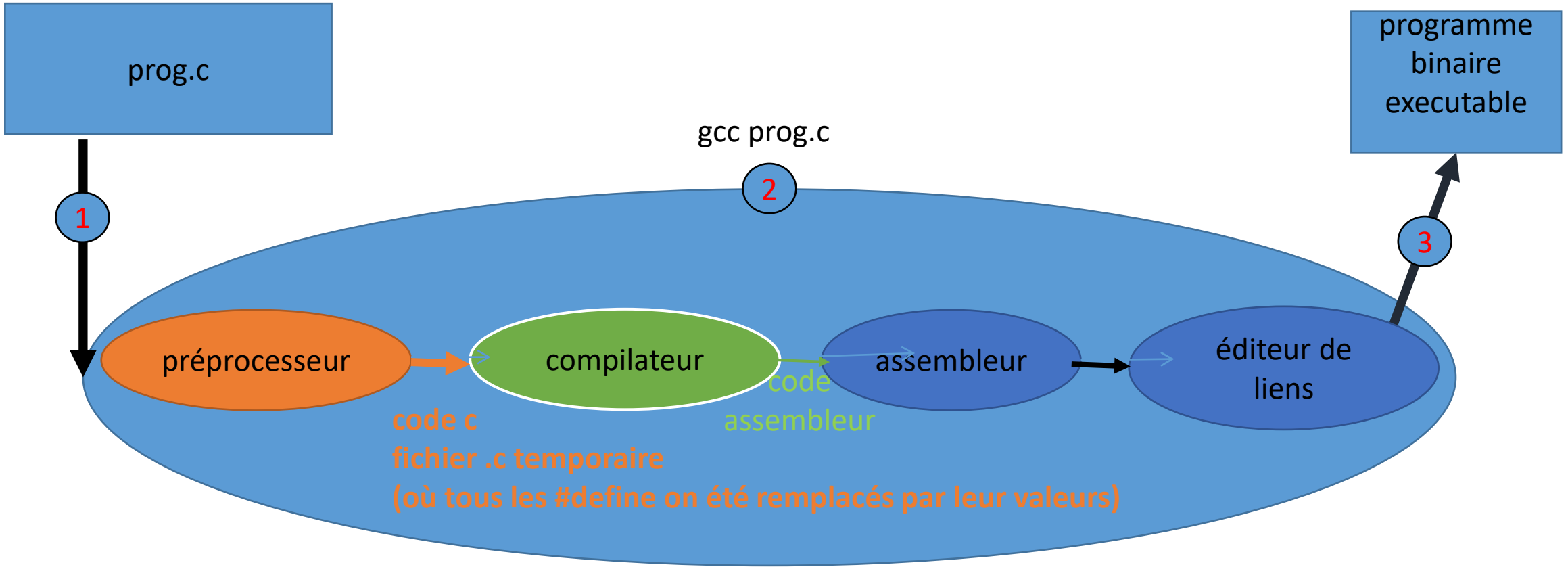


Le préprocesseur va
1) chercher MAX
2) remplacer par 3

permet de ne modifier le programme qu'à un seul endroit

Les tableaux : #define

- Chaine de compilation (détaillée dans le dernier chapitre) :



Les tableaux : deux dimensions (2D)

- Déclaration : **type NomTab[NbEltDimension1][NbEltDimension2];**

  NbrLignes NbrColonnes

  (les éléments de chaque ligne sont contigües en mémoire)

- Initialisation possible uniquement lors de la déclaration :

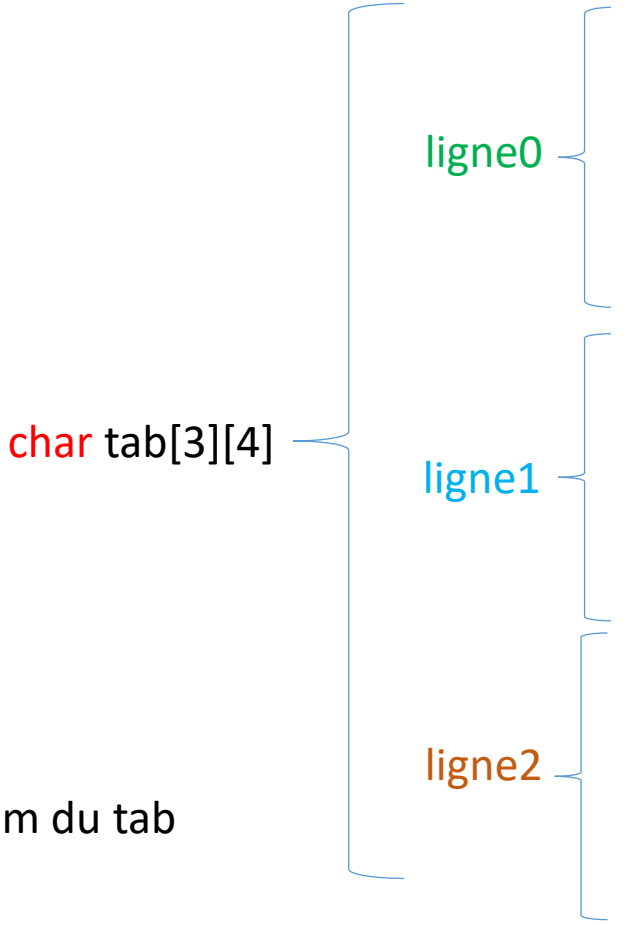
```
int main () {
        char tab1[3][4];
        int tab2[3][4]={ {1,2,3,4} , {5,6,7,8} , {9,10,11,12} };
        ….
```

- Indices : ils partent toujours de 0

	col. 0	col. 1	col. 2	col. 3
ligne 0	1	2	3	4
ligne 1	5	6	7	8
ligne 2	9	10	11	12

Tab2 [0][0] =1	Tab2 [0][1] =2	Tab2 [0][2] =3	Tab2 [0][3] =4
Tab2[1] [0] = 5	Tab2[1] [1] = 6	Tab2 [1][2] =7	Tab2 [1][3] =8
Tab2 [2][0] =9	Tab2 [2][1] =10	Tab2 [2][3] =11	Tab2 [2][3] =12

Les tableaux 2D : en mémoire



		Adresses	Contenu
col. 0	**tab[0][0]**	90	xxx
col. 1	**tab[0][1]**	91	xxx
col. 2	**tab[0][2]**	92	1
col. 3	**tab[0][3]**	93	2
col. 0	**tab[1][0]**	94	3
col. 1	**tab[1][1]**	95	4
col. 2	**tab[1][2]**	96	5
col. 3	**tab[1][3]**	97	6
col. 0	**tab[2][0]**	98	7
col. 1	**tab[2][1]**	99	8
col. 2	**tab[2][2]**	100	9
col. 3	**tab[2][3]**	101	10
		102	11
		103	12
		104	xxx
		105	xxx

Rappel : l'@ du 1er élément = nom du tab
tab ⇔ &tab[0][0]

Ici tableau de char : 1 octet = 1 élément…

Les tableaux 2D : en mémoire

- Comment le compilateur calcul l'@ de tab[i][j] ?
- Il faut :
  - l'adresse de base du tableau
  - ET la taille d'une ligne➔(nombre d'élément d'une ligne (= nombre_col)* taille d'1 élément

&tab[i][j] ⇔ &tab[0][0] + i*taille_ligne +j;

Exemple :

char tab[3][4]

&tab[1][3] ?

92 + 1 * 4+ 3 = 92+4+3= 99

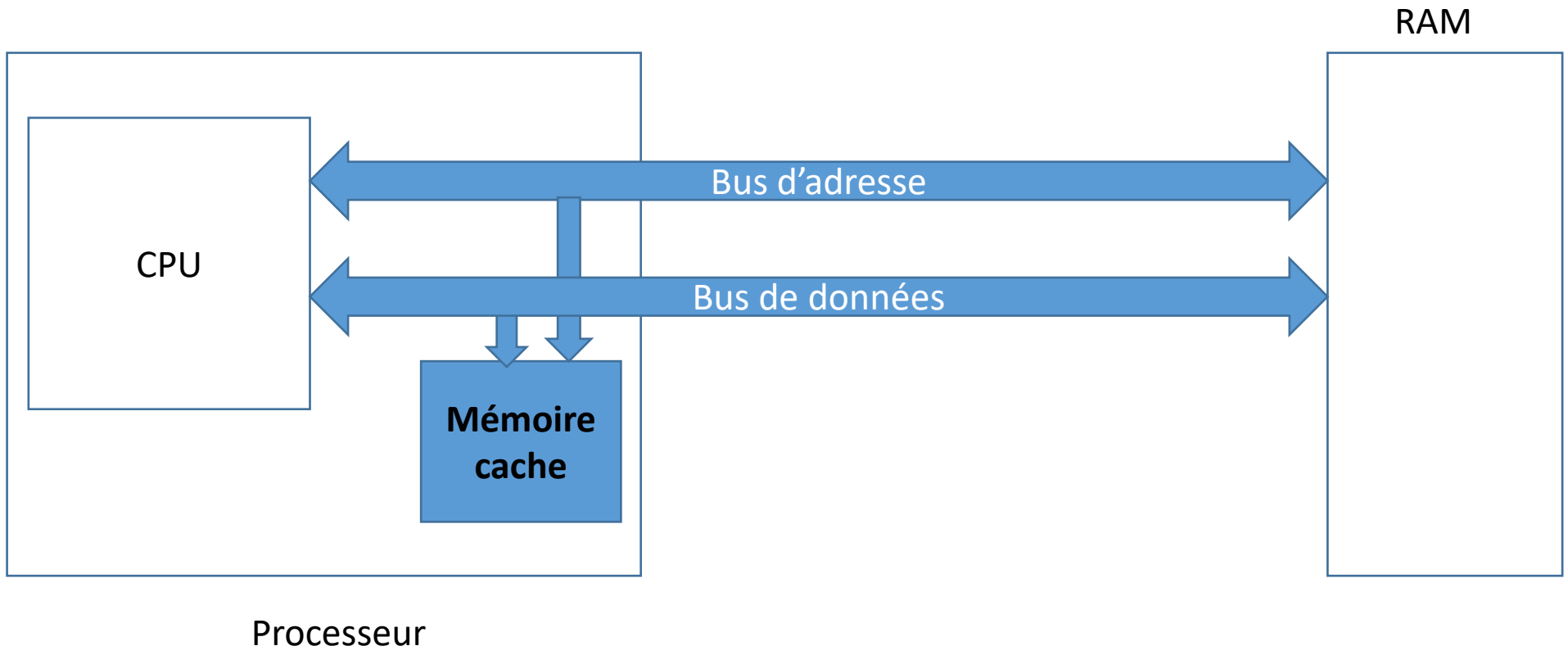
		Adresses	Contenu
		90	xxx
		91	xxx
col. 0	tab[0][0]	92	1
col. 1	tab[0][1]	93	2
col. 2	tab[0][2]	94	3
col. 3	tab[0][3]	95	4
col. 0	tab[1][0]	96	5
col. 1	tab[1][1]	97	6
col. 2	tab[1][2]	98	7
col. 3	tab[1][3]	99	8
col. 0	tab[2][0]	100	9
col. 1	tab[2][1]	101	10
col. 2	tab[2][2]	102	11
col. 3	tab[2][3]	103	12
		104	xxx
		105	xxx

1) Le nombre de ligne n'est pas nécessaire pour calculer les @ des éléments
2) Un tableau 2D se ramène à un tableau 1D

Les tableaux 2D : en mémoire

- Intérêt de comprendre ?
  - Ecrire du code plus simple et plus efficace : 1 seule boucle pour parcourir tout un tableau 2D (pour l'initialiser par exemple)
  - Eviter les défauts de mémoire cache (pour les tableaux qui ne tiennent pas en mémoire cache) : en effectuant des accès de colonne en colonne plutôt que de ligne en ligne
  - Comprendre et éviter des bugs...

Les tableaux 2D : mémoire externe et mémoire cache



Les tableaux : N dimensions

- Déclaration : **type NomTab[NbEltDim1] [NbEltDim2] [NbEltDim3] ... ;**

Types et structures de données du C

- Types entiers
  - char / unsigned char
  - affichage en hexa, opérateur sizeof, codage ASCII
  - short, endianness
  - int
- Type flottants
  - Limites du codage en virgule fixe
  - Codage IEEE 754 (float, double)
  - Avantage inconvénients : FPU…
- Tableaux
  - 1D, déclaration, initialisation, allocation mémoire, erreurs fréquentes
  - 1D Chaîne de caractères, bibliothèque string.h
  - 2D
- **Structures**
- Les pointeurs (1ère partie)

Les Structures

- Tableaux : regroupent éléments de même type
- Structures : regroupent des éléments de types différents
- Déclaration :

optionnel

```
struct NomDeLaStructure {
   type_donnée_champs1    NomDuChamps1;
   type_donnée_champs2    NomDuChamps2;
   etc...
}  NomDeVariable ;
```

optionnel

Exemple :

```
struct Pixel {
  int x;
  int y;
};

puis allocation :
struct Pixel p1;
```

- Pour "créer une instance de cette structure" (allocation mémoire) :

  struct **NomDeLaStructure** **NomVariable**;

  ➔ c'est à ce moment qu'un espace mémoire est réservé, pas avant

Structure : accès aux champs

- Pour accéder ou modifier à champs on utilise :
  - le nom de la variable de la structure
  - de l'opérateur unaire point '.'
  - du nom du champs à lire/modifier

  - Exemple :
    - NomVariable.NomChamps1 = valeur1;
    - valeur2 = NomVariable.NomChamps2;

Exemple :

```
struct Pixel {
   int x;
   int y;
};
```

Allocation :

```
struct Pixel p1;
```

Modification :

```
p1.x = 10;
printf("champs y=%d", p1.y);
```

Structures : initialisation lors de l'allocation

- Il est possible d'allouer la mémoire et initialiser les champs de la structure en une seule fois (uniquement lors de la déclaration de la variable, plus ensuite) :

Exemple : si la structure Pixel à été défini précédemment:

- struct Pixel p1 = { 10, 20}; ➔ le compilateur utilise l'ordre de déclaration des champs (ici x puis y), si il manque des valeurs : les champs seront init. à 0.

- en C99 on peut les initialiser en les nommant :
  - struct Pixel P1 = {.y=20, .x = 10};

Structure : exemples complets

```
#include<stdio.h>
struct Pixel {
   int x;
   int y;
};
int main() {
      struct Pixel p1 = {10, 10} ;
      p1.x = 55;
      p1.y = 22;
      printf(" p1.x = %d et p1.y = %d", p1.x,
         p1.y);
      return 0;
}
```

```
#include<stdio.h>
int main() {
   struct Pixel {
            int x;
            int y;
   } p1;
   p1.x = 10;
   p1.y = 20;
   printf(" p1.x = %d et p1.y = %d", p1.x,
   p1.y);
   return 0;
}
```

optionnel, si l'on a pas besoin d'en créer d'autres

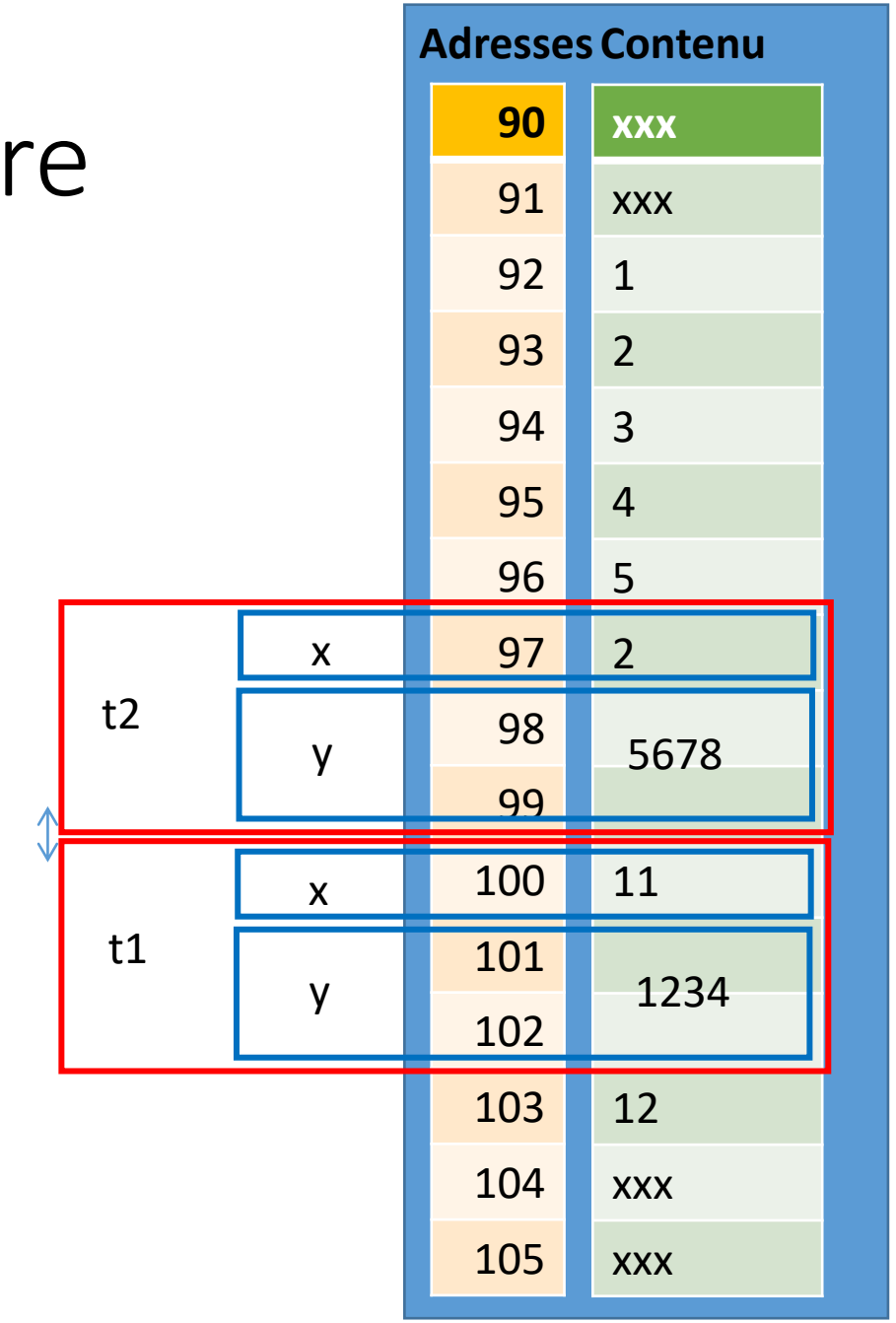
Structure : occupation mémoire

Exemple :

```
struct Test {
  char x;
  short y;
} ;
```

(pas forcément contigües)

```
struct Test t1 = {11,1234}, t2 = {2,5678};
 sizeof(t1) ?
 sizeof(struct Test) ?
```



Structure incluant des tableaux

- Les champs d'une structure peuvent être des tableaux, d'autres structures...:
  - Exemple :

```
struct Eleve {
    char Nom[30];
    char Prenom[30];
    unsigned char age;
};
```

  - Initialisation :
    - struct Eleve E1={ "Kenobi", "ObiWan", 30} ;
  - Accès :
    - printf("le prenom de E1 est %s et son age est %d ", ...};
  - Modification :
    - Pour changer l’age :
    - Pour changer le nom :
    - Pour changer le prénom :

Copier une structure

struct Eleve E1={ "Kenobi", "ObiWan", 30} ;

struct Eleve E2;

E2=E1; ➔ tous les champs de E1 sont recopiés dans E2

printf(‘’Nom dans E2 = %s’’, E2.nom); ➔ affiche ……..

Tableaux de structures

- On peut allouer des tableaux de structures : chaque élément est une structure :
  - Exemple :

    struct Pixel TabPixels[10];

- L'initialisation se fait comme pour les tableaux :
  - struct Pixel TabPixels[10] = { {10,10}, {12,11}, {5,2} } ;

➔ comment stocker x=33 et y=66 à l'indice 4 par exemple ?

Structure incluant des tableaux

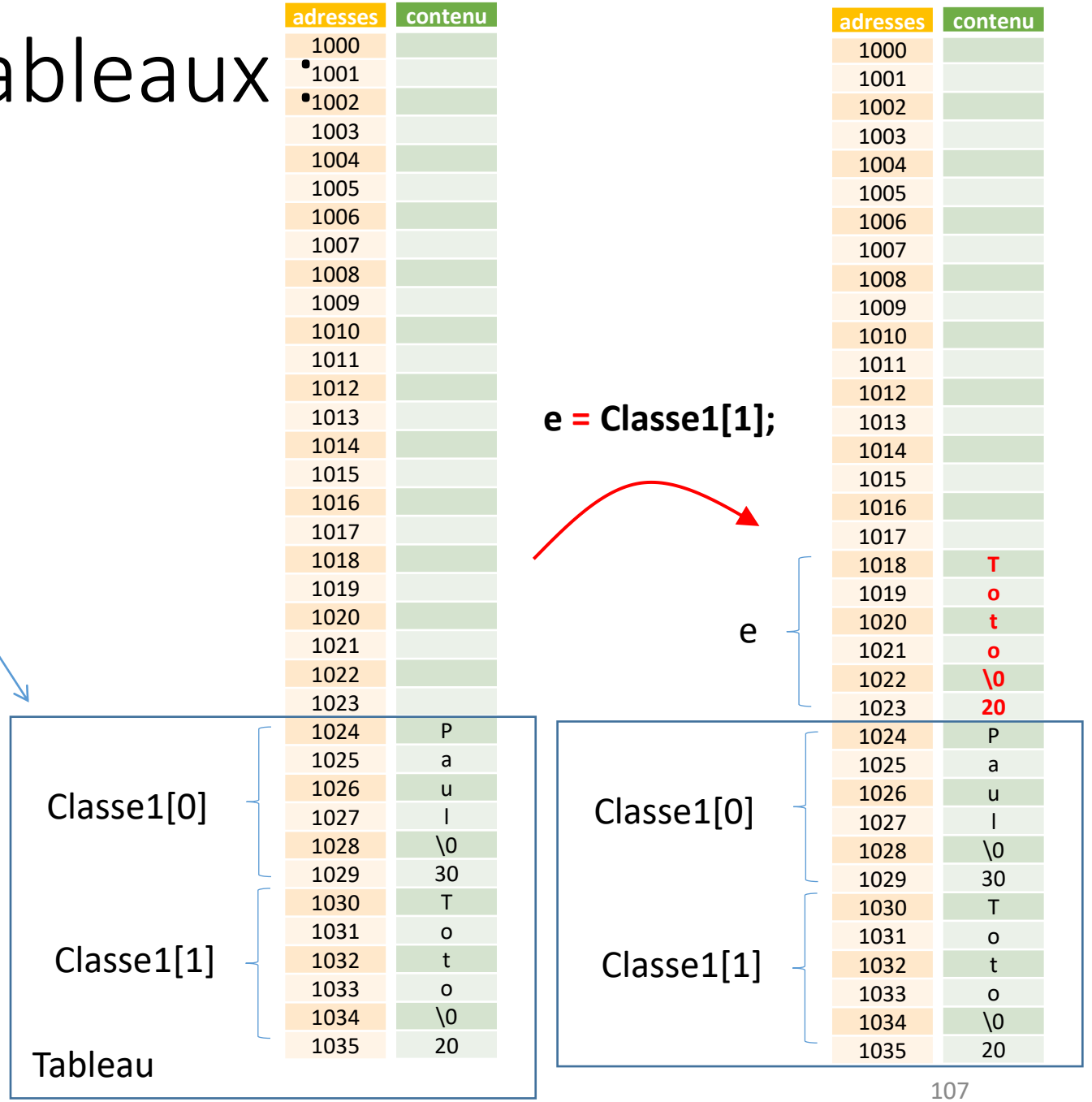
- Les champs d'une structure peuvent être des tableaux, d'autres structures...:
  - Exemple :

```
struct Eleve {
    char Nom[30];
    char Prenom[30];
    unsigned char age;
};
```

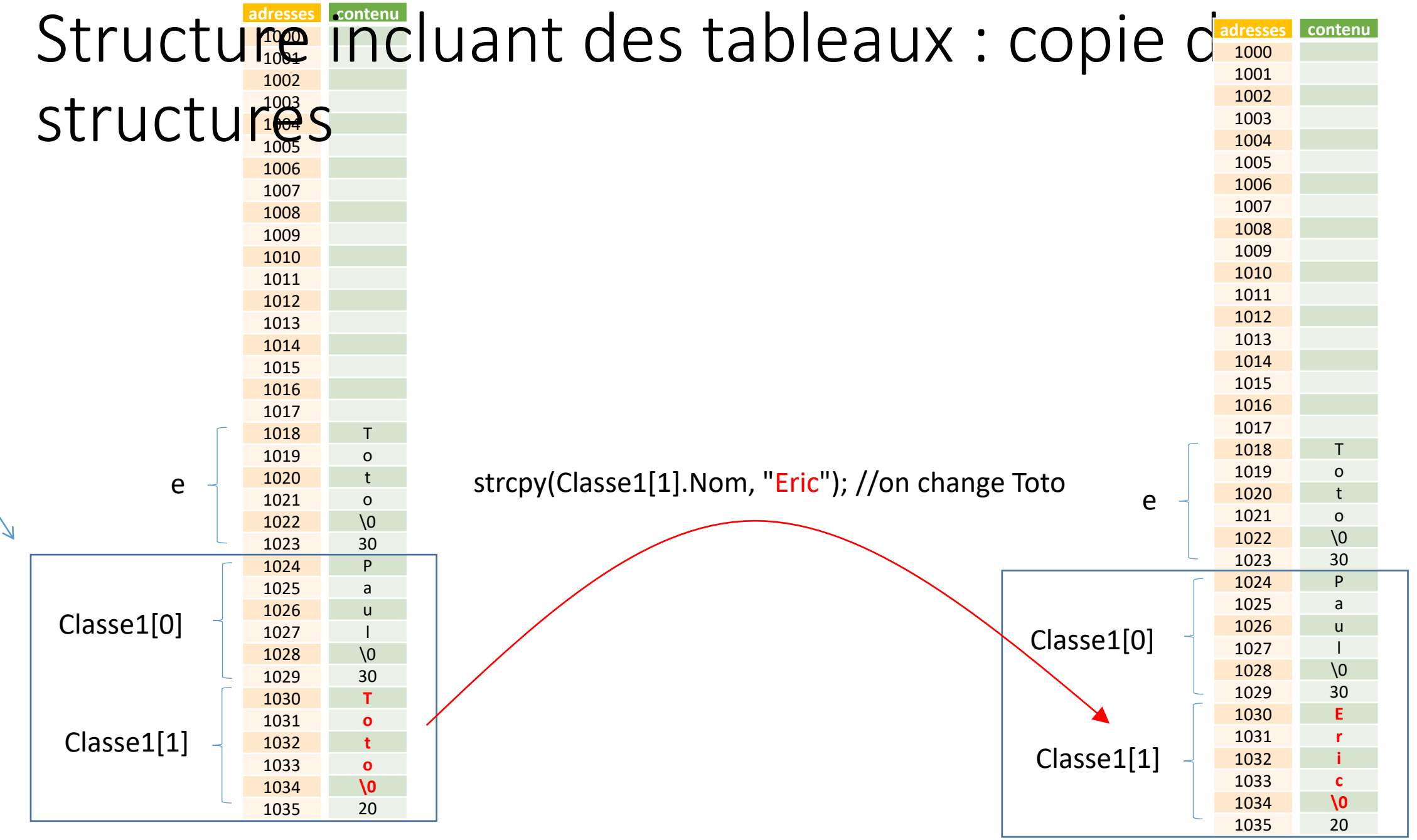
  - Initialisation :
    - struct Eleve Classe1[30]={ {"Kenobi", "ObiWan", 30} , {"Lannister", "Cersei", 20} };
  - Accès :
    - printf("les noms des 2 premiers du tableau sont %s et %s ", ...};
  - Modifications :
    - Changer l'age du 3eme ?
    - Changer le nom du 1er ?

Structure incluant des tableaux : copie de structures

```
#include <stdio.h>
struct Eleve {
  char Nom[5]
unsigned char age;
};
int main() {
  struct Eleve Classe1[2]={ {"Paul", 30} , {"Toto", 20} };
  struct Eleve e;
  e = Classe1[1];
  strcpy(Classe1[1].Nom, "Eric"); //on change Toto
  printf("Nom de Classe1[0] : %s, \n", Classe1[0].Nom);
  printf("Nom de e : %s", e.Nom);
 return 0;
}
```



Structure incluant des tableaux : copie d structures



Les unions : définition

- Dans le cas des structures, les champs ne partagent pas les mêmes zones mémoire : ils sont indépendants
- Dans une unions, les champs partagent le même emplacement mémoire !
  - modifier le contenu d'un champs affecte les autres champs

optionnel

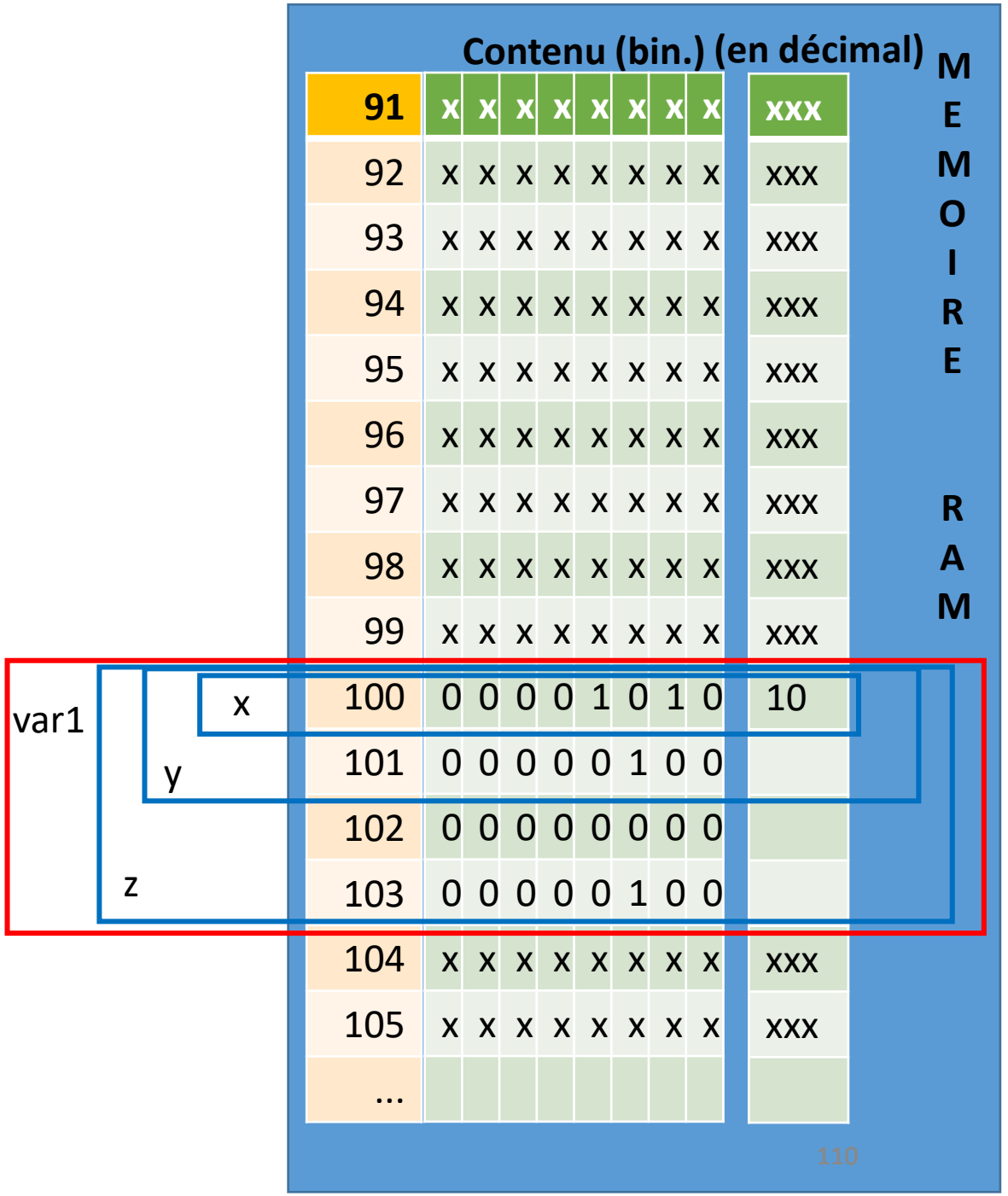
Déclaration :

```
union NomUnion {
   type_donnée_champs1    NomDuChamps1;
   type_donnée_champs2    NomDuChamps2;
   etc...
}  NomDeVariableUnion ;
```

optionnel

Les unions : exemple

```
#include<stdio.h>
union Union1 {
   char x;
   short y;
   int z;
};
int main() {
      union Union1 var1 ;
      var1.z = 67109898; /*0x0400040A */
      printf(" var1.x = %d , var1.y = %d, var1.z= %d", var1.x,
         var1.y, var1.z);
      return 0;
}
```



Types et structures de données du C

- Types entiers
  - char / unsigned char
  - affichage en hexa, opérateur sizeof, codage ASCII
  - short, endianness
  - int
- Type flottants
  - Limites du codage en virgule fixe
  - Codage IEEE 754 (float, double)
  - Avantage inconvénients : FPU…
- Tableaux
  - 1D, déclaration, initialisation, allocation mémoire, erreurs fréquentes
  - 1D Chaîne de caractères, bibliothèque string.h
  - 2D
- Structures
- **Les pointeurs (1ère partie)**

Les pointeurs

Rappel :

- chaque variable (char, short, int, struct, tableau...) est stocké en mémoire et possède donc une adresse en mémoire.
- on obtient cette adresse avec & : exemple printf("%p", &z);

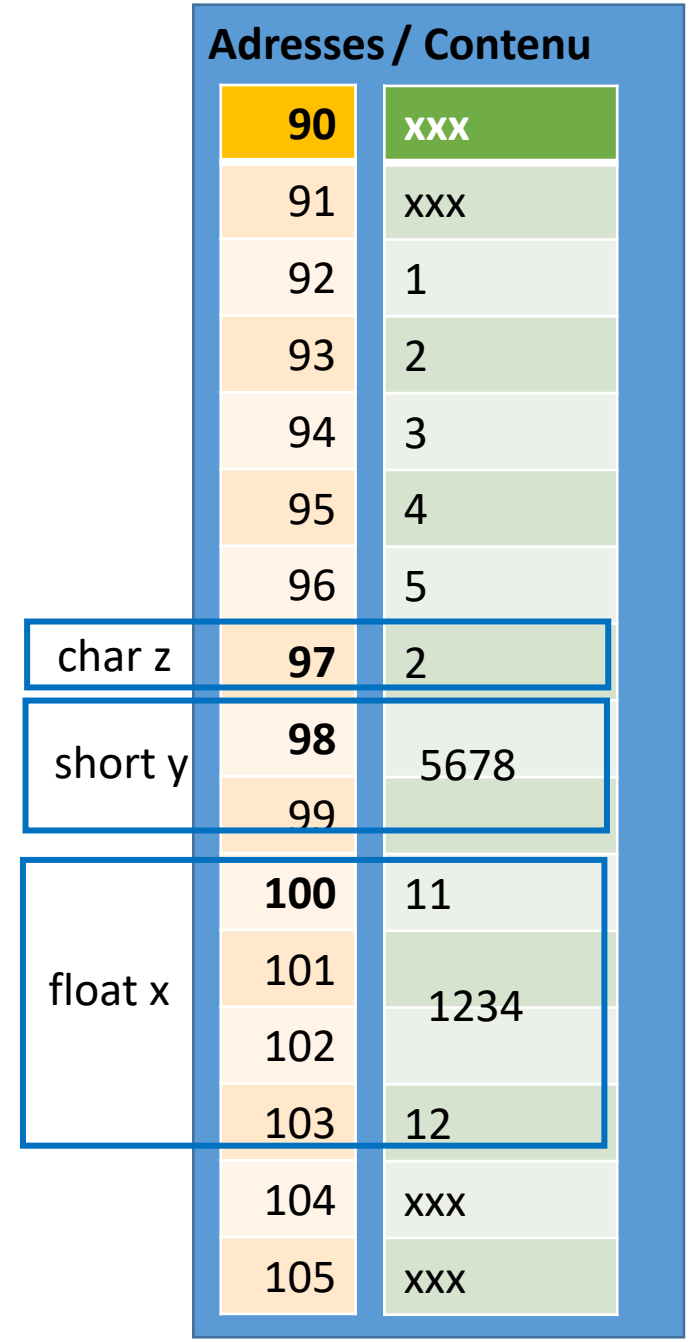
Quelle est la taille d'une adresse ?

Dans quoi stocker une adresse ?

Dans une variable particulière appelée pointeur.

Un pointeur est une variable qui contient l'adresse d'une autre variable.

Quelle est la taille d'un pointeur ? La même qu'une adresse ! mais encore...



Pointeurs

Un pointeur contient forcément un entier positif ou NULL.

Un pointeur contient l'adresse d'une variable.

Pour pouvoir interpréter correctement la valeur d'un pointeur il faut savoir de quelle type est la variable dont il contient l'adresse.

C'est à dire qu'il faut indiquer vers quel type de donnée il va pointer : vers un int ? vers un char ? vers un tableau de float ?...

Lorsque l'on déclare un pointeur, on indique donc vers quoi il va pointer.

Pour déclarer un pointeur on utilise le signe étoile * entre le nom de type et le nom de la variable pointeur (on dira pointeur) :

type * NomVarPointeur;

Pointeurs : déclaration

- Si la variable p1 doit contenir l'adr. d'un int on déclare le pointeur par :

  int *p1;

- Si le pointeur p2 doit contenir l'adresse d'un char on le déclare par :

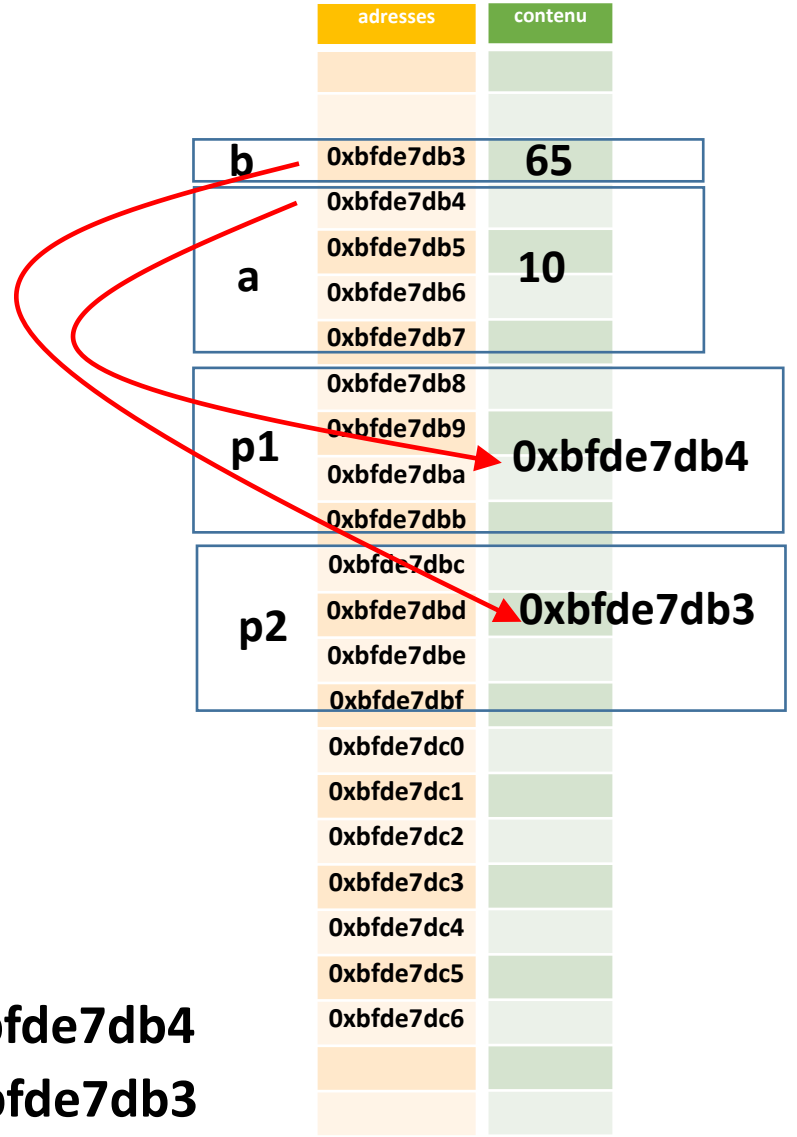
  char *p2;

- Si le pointeur p3 doit contenir l'adresse d'une structure Pixel déjà défini on le déclare par :

  struct Pixel *p3;

Pointeurs : affectation

```
#include <stdio.h>

int main () {
    int a=10;
    char b='A';
    int *p1;
    char *p2;

    p1=&a;  //affectation : maintenant p1 contient l'@ de a,
                          on dit que "p1 pointe vers a"
    p2=&b;  // p2 pointe vers b

    printf("adresse de a : %p \n", p1);
    printf("adresse de b : %p \n", p2);
    printf("Adresse de p1 : %p", &p1);
    printf("taille de p1 : %d", sizeof(p1));
    return 0;
```



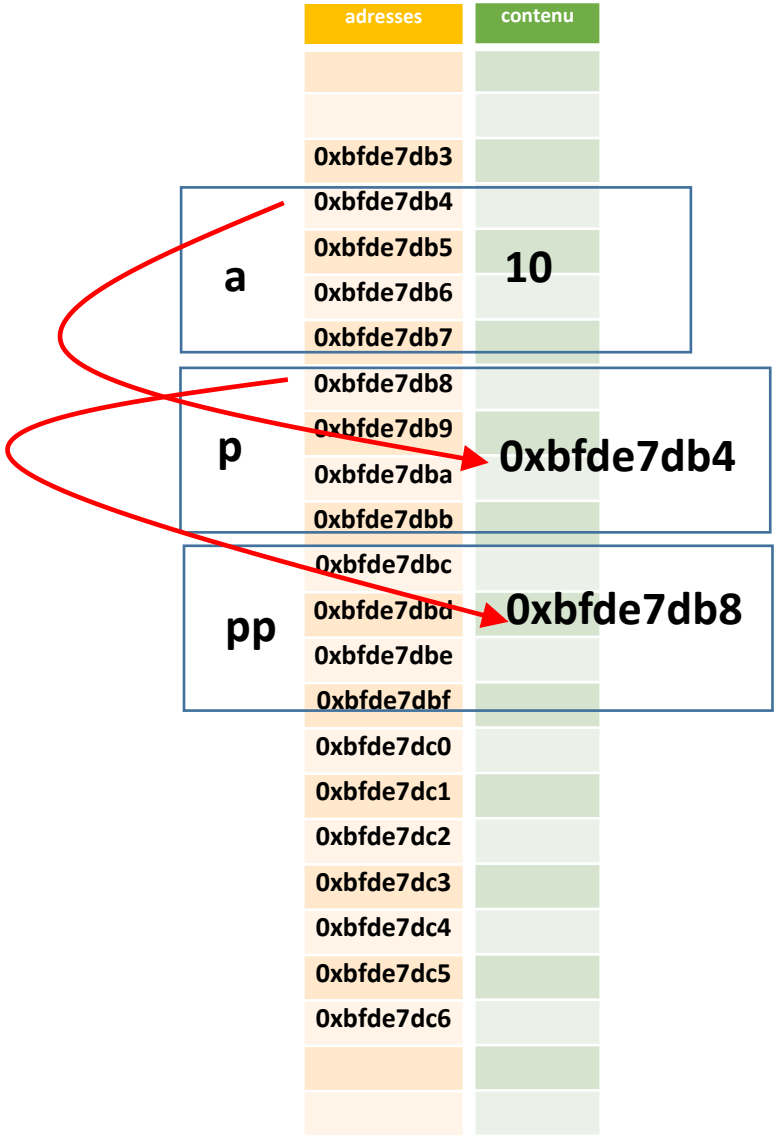
Exécution :

adresse de a : 0xbfde7db4
adresse de b : 0xbfde7db3
Adresse de p1 : 0xbfde7db8
taille de p1 : 4

Pointeur vers un ...pointeur

Dans quoi stocker l'adresse d'un pointeur ?
Dans un pointeur de pointeur à l'aide de 2 * :

```
#include <stdio.h>
int main () {
  int a=10;
  int *p;
  int **pp;
  p=&a;
  pp=&p;
  printf("adresse de p = %p \n", &p);
  printf("adresse de p = %p \n", pp);
  return 0;
}
```



Pointeurs : taille

```
#include <stdio.h>
int main () {
    int a=10;
    char b='A';

    int *p1;
    char *p2;

    p1=&a;  //affectation : maintenant p1 contient l'@ de a, "p1 pointe vers a"
    p2=&b;  // p2 pointe vers b

    printf("sizeof(p1) = %d \n", sizeof(p1));
    printf("sizeof(p2) = %d \n", sizeof (p2) );

return 0;
}
```

Tous les pointeurs occupent la même taille.

Mais les objets vers lequel chacun pointe sont de tailles différentes

p1 pointe vers un int

p2 pointe vers un char

p1= (int *)&b; ➔ « caster »

Pointeurs : cast

Soit

```
int *p1;
char b;
```

Si l'on écrit :

```
p1= &b;
```

➔Warning car les types sont différents

Pour l'éviter, on doit « caster »

```
p1= (int *) &b;
```

Pointeur : accès et modif. d'une var. pointée

Si un pointeur pointe vers une variable (⇔ il contient l'@ de cette var.) **alors** on peut lire (et modifier) le contenu de cette variable avec l'opérateur unaire * placé devant le nom du pointeur :

*pointeur

```
int a=10;
int *p;
p=&a;
printf("%d", *p);   affiche la val. de a car p pointe vers a
*p = 20;            modifie la val. de a car p pointe vers a
```

Principe identique à l'accès en assembleur avec :

LDR R2, [R1] avec R1 contenant une adresse mémoire

Pointeur : accès et modif. d'une var. pointée

Si p est un pointeur initialisé (il pointe vers une variable ⇔ il contient l'adresse d'une variable)

*p = 20 ; ⇔ je stocke la valeur 20 à l'adresse stockée/indiquée dans p

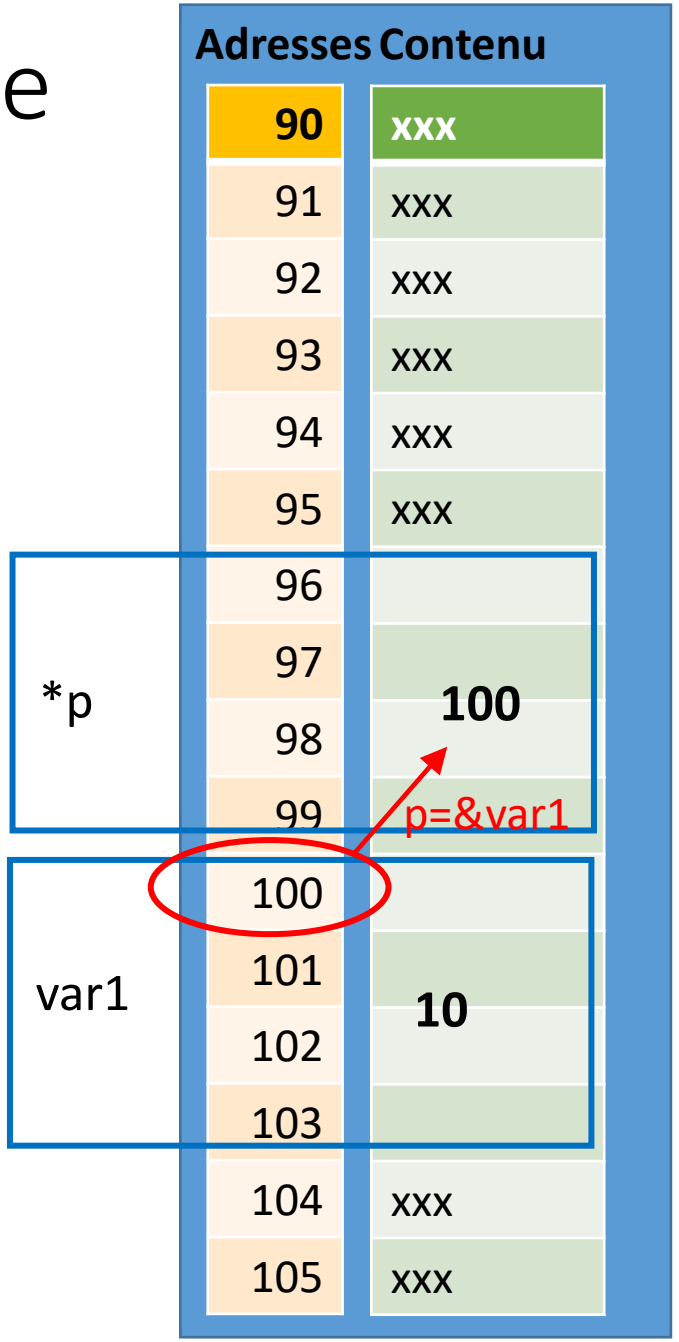
⇔ je stocke la valeur 20 à l'adresse pointée par p

Le type de la donnée pointée doit être de même type que le pointeur (sinon le compilateur ne peut savoir combien de cases il faut copier/lire)

Pointeur : accès et modif. d'une var. pointée

Si p est un pointeur vers une variable *p permet d'accéder à cette variable pour la lire ou l'écrire

```
#include <stdio.h>
int main () {
  int var1=10;
  int *p;
  p=&var1;
  printf("La variable pointée par p (cad var1) vaut %d", *p);
  ⇔  printf("La variable pointée par p (cad var1) vaut %d", var1);

  *p = 20;  // ce qu'il y a à l'@ contenue dans (=pointée par) p reçoit 20
            //⇔ var1=20;  car p pointe vers var1

   printf("La variable var1 vaut maintenant : %d", var1);
   return 0;
}
```



Pointeurs : intérêt ?

- Permet d'accéder aux périphériques du processeur
- Permet d'accéder librement à la mémoire
- Permet d'implanter des structures de données complexes
- Permet de faire des modifications (lectures, écritures) et construire des fonctions impossibles sans eux (par exemple les fonctions ne permettent de retourner qu'une seule valeur....)

Pointeur : lecture d'une adresse mémoire

Exemple :

```
int b,*p;
p=12000;
b=*p;
printf(‘’le contenu de l’adresse %p est %d’’, p, b);
```

Ou sans utiliser de pointeur :

```
int b;
b=*(12000);    /*Pb. : provoque un warning car ne connait pas le type pointé */
```

Plus correctement pour éviter le warning on doit caster :

```
b=*(int *) (12000);
```

Pointeurs : accès aux périphériques (Evalbot E1)

```
int main(void) {
 int a=0;
 // LED sur broche PF5, p291 : GPIOF = bit 5 = 0x20
 *(int *) (0x400FE000+0x108)= *(int *)(0x400FE000+0x108)| 0x20;
    for (a=0;a<10;a++);      //delai de 3 clock
 //GPIO_PORTF_BASE=0x40025000   et   GPIO_PIN_5=0x00000020
 // Puissances de sortie 2mA => activer bit dans GPIO_O_DR2R, les autres à
    0
 //GPIO_O_DR2R  @= 0x00000500  // GPIO 2-mA Drive Select ,p214
    *(volatile unsigned long *)(0x40025000+0x00000500)=  0x00000020;
 //GPIO_O_DEN  @= 0x0000051C  // GPIO Digital Enable
    *(volatile unsigned long *)(0x40025000+0x0000051C)=  0x00000020;
 //Direction des ports : out , GPIO_O_DIR   @=  0x00000400
    *(volatile unsigned long *)(0x40025000+0x400) = 0x00000020;
 while(1) {
    //Ecriture sur le port
     // Offset (0x20<<2) : port 0x20 = broche 5

    *(int *) (0x40025000+(0x20<<2)) = 0x00;

    for (a=0;a<1000000;a++)
          ; // boucle attente visuelle

    *(int *) (0x40025000+(0x20<<2)) = 0x20;

    for (a=0;a<1000000;a++)
          ;  //boucle attente visuelle
    }
}
```

Pointeurs : intérêt ?

- Suite dans le chapitre 5 : pointeurs avancés

(pointeurs et fonctions, pointeurs et tableaux)

IV- Opérateurs, Structures de Contrôle, Fonctions

Ce cours s'adresse à des élèves initiés à Java en E1, nous ne développerons donc pas les parties communes aux 2 langages.

Opérateurs bit à bit : ~ & | ^ >> <<

NON : ~

ET : &

OU : | (touche 6 clavier)

OU EXCLUSIF : ^

Décalage droite : var**>>**nb_bits

Décalage gauche : var**<<**nb_bits

Ils insèrent des 0 et perdent les bits

=> ce ne sont pas des rotations

Utile pour configuration

du matériel : registres périphériques

Exemple :

```
unsigned char f,e,d;
unsigned char c = 15;  //0x0F
unsigned char b = 240; //0xF0
unsigned char a = 170; // 0xAA

d = a & b;
e = ~a;
f = c <<1;

printf("a=%x, b=%x, c=%x \n",a,b,c);
printf("d=a & b = %x \n", d);
printf("e =~a = %x \n", e);
printf("f = c<<1 = %x", f);
```

	x	x	x	x	x	x	x	x	xxx
	x	x	x	x	x	x	x	x	xxx
	x	x	x	x	x	x	x	x	xxx
a	1	0	1	0	1	0	1	0	170
b	1	1	1	1	0	0	0	0	240
c	0	0	0	0	1	1	1	1	15
d	1	0	1	0	0	0	0	0	160
e	0	1	0	1	0	1	0	1	85
f	0	0	0	1	1	1	1	0	30
	x	x	x	x	x	x	x	x	xxx
	x	x	x	x	x	x	x	x	xxx
	x	x	x	x	x	x	x	x	xxx
	x	x	x	x	x	x	x	x	xxx
	x	x	x	x	x	x	x	x	xxx
	x	x	x	x	x	x	x	x	xxx

Opérateurs de comparaisons

Test d'égalité : ==

Test de différence : !=

Inférieur, supérieur : < , >

inférieur ou égal : <= (attention =< est faux)

sup. ou égal : >= (attention => est faux)

Ne pas confondre test d'égalité et comparaison :

if (a == b) : compare les valeurs de a et b

if (a=b) : copie la valeur de b dans a puis TEST si a est vrai (cad ≠ 0)

Opérateurs logiques : && || !

Pour comparer 2 expressions booléennes (chacune =0 ou ≠0)

&& : ET logique

|| : OU logique

! : NON logique

Exemple :

if ((x<0) || (x>100)) .... //ne pas confondre avec OU bit à bit

Opérateurs arithmétiques : + - * / %

Attention aux différences entre i++; et ++i;

Exemple :

```
int a=10, b=0;
printf (" a= %d, b=%d \n", a,b);
b=a++;
printf (" a= %d, b=%d \n", a,b);
```

```
int a=10, b=0;
printf (" a= %d, b=%d \n", a,b);
b=++a;
printf (" a= %d, b=%d \n", a,b);
```

A la fin, b n'a pas la même valeurs dans ces 2 exemples

Opérateurs arithmétiques : + - * / %

Utilisation du signe égale adossé à un opérateur mathématique :

Opérateurs	Exemples
+=	a+=3; ⇔ a=a+3;
-=	a-=5; ⇔ a= a-5;
*=	a*=2; ⇔ a=a*2;
/=	a/=2; ⇔ a=a/2;

Structure de contrôle : if ...else, else if

```
if (expression) {
   instruction1;
   instructionN ;
}
```

```
if (expression) {
   instruction1;
   instructionN ;
} else {
   instruction1;
   instructionN;
}
```

expression : si sa valeur est ≠ 0
➔ considéré comme vrai
si l'expression =0
➔ considéré comme faux

```
if (expression1) {
   instruction1;
   instructionN ;
} else if ( expression 2) {
   instruction1;
   instructionN;
} else if ( expression 3) {
   instruction1;
   instructionN;
}
```

TOUJOURS INDENTER APRES CHAQUE ACCOLADE (= tabulation, espace)

Structure de contrôle : if ...else, else if

```
if (expression1)
   instruction;
else if ( expression 2)
   instruction;
else if ( expression 3)
   instruction;
```

Si une seule instruction à exécuter dans le test, pas besoin d'accolades

...mais ne pas oublier d'indenter!

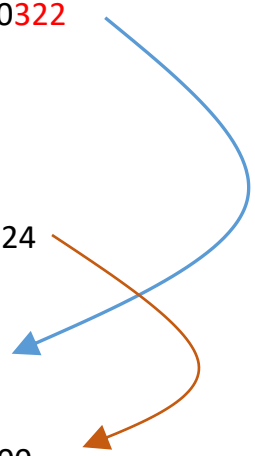
Facilite fortement la lecture du code !!

Structure de contrôle : if ...else en ASM

```
int main(void) {
   int a=10,b=20,c=0;
   if (a<0)
        c=a;
   else
        c=b;
}
```

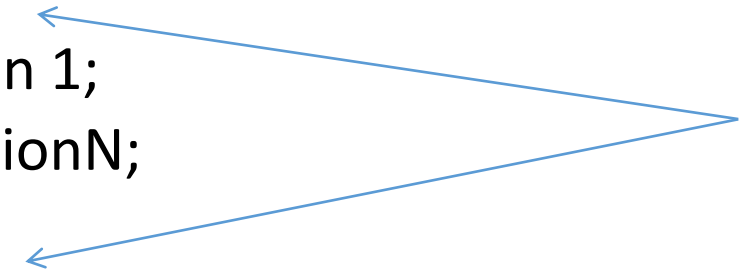
Code généré en assembleur ARM Cortex M :

```
7:    int a=10,b=20,c=0;
0x00000314 210A     MOVS    r1,#0x0A
0x00000316 2214     MOVS    r2,#0x14
0x00000318 2300     MOVS    r3,#0x00
   8:    if (a<0)
0x0000031A 2900     CMP     r1,#0x00
0x0000031C DA01     BGE     0x00000322
   9:        c=a;
  10:        else
0x0000031E 460B     MOV     r3,r1
0x00000320 E000     B       0x00000324
  11:        c=b;
0x00000322 4613     MOV     r3,r2
  12: }
0x00000324 2000     MOVS    r0,#0x00
0x00000326 4770     BX      lr
```



Structure de contrôle : switch ...case

```
switch (expression) {
   case valeur1 : instruction 1;
                  instructionN;
                  break;
   case valeur2 : instruction 1;
                  instructionN;
                  break;
   default :
                  instruction1;
                  instructionN;
}
```



break optionnel !
dépend de ce que l'on veut faire
(mais en général indispensable)

Attention : uniquement pour des entiers

Structure de contrôle : boucles for

```
for (inst.d'initialisation; condition_pour_continuer; inst._de_modification_à_chaque_itération) {
   instruction1;    (si 1 seule instruction accolades inutiles)
   instructionN;
}
```

1. La première fois exécute les instructions d'initialisation
   (souvent 1 seule, l'initialisation du compteur de boucle
2. Tant que la condition est vrai (cad ≠ 0) on exécute :
   1. les instructions d'itération (souvent 1 seule : l'incrément du compteur)
   2. puis le corps de la boucle

Dans la première et la dernière partie il peut y avoir plusieurs instructions séparées par des virgules

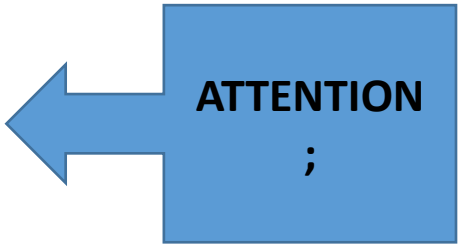
A TOUT MOMENT ON PEUT SORTIR D'UNE BOUCLE AVEC L'INSTRUCTION break

Structure de contrôle : while, do...while

```
while (expression) {
   instruction1;
   instructionN;
}
```

1) On test l'expression
2) Tant que l'expression est est vrai (cad ≠ 0) : on execute le corps de la boucle

```
do {
   instruction1;
   instructionN;
} while (expression);
```



1) On exécute le corps de la boucle
2) Tant que l'expression est vrai on recommence en 1)

A TOUT MOMENT ON PEUT SORTIR D'UNE BOUCLE AVEC L'INSTRUCTION *break*

Fonctions : définition

- Une fonction est définie par :
  - un nom
  - la liste de ces arguments
  - le type retourné par la fonction (void si retourne rien, l'invocation du mot clef **return** fait alors sortir de la fonction), la valeur sera retourné par le mot clef **return NomVar;**
  - et son corps

```
type_retourné NomFonction ( TypeArg1 NomVarArg1, TypeArg2 NomVarArg2, etc) {
    Corps_de_la_fonction
    return valeur;  //optionnel si la fn ne retourne rien, ou encore return;
}
```

Attention : le nom doit être unique dans tous le programme (≠ JAVA où on a 1 namespace/class) et même si le nombre d'arguments diffère (ds ce cas donner plusieurs nom différents)

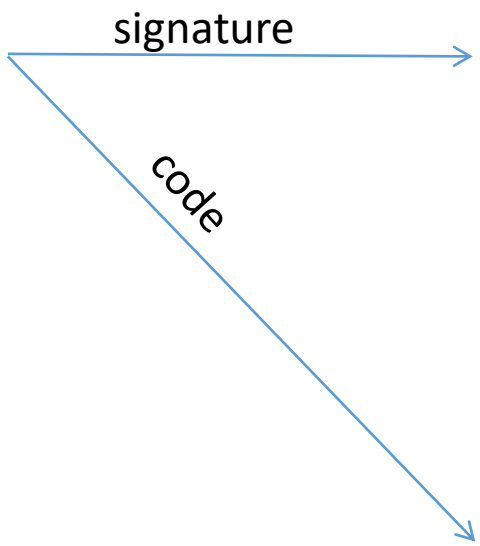
Le compilateur doit connaître le prototype (=signature) de la fonction avant de l'utiliser :

- soit déclarer la fonction avant de l'utiliser
- soit déclarer uniquement sa signature suivie avant de l'utiliser. Le code de la fonction peut alors être donné après la fonction qui l'utiliser

Fonctions : exemple sans arg. et val. de retour

Déclaration avant utilisation :

```
#include <stdio.h>
void AfficheBonjour( void) {
   printf("Bonjour");
}

int main () {
   AfficheBonjour  ();
    return 0;
}
```



Code après utilisation

```
#include <stdio.h>
void AfficheBonjour( void);

void main () {
   AfficheBonjour  ();
}

void AfficheBonjour( void) {
   printf("Bonjour");
}
```

Exemple 2 : exemple arg. et val. de retour

```
#include <stdio.h>
float max3 (float a, float b, float c);

int main () {
   float x=1.0,y=2.0,z=3.0,res=0.0;
   res = max3(x,y,z);
   printf("max(%f,%f,%f)=%f",x,y,z,res)
   ;
```

(suite)

```
float max3 (float a, float b, float c) {
   if ( (a>=b) && (a>=c) )
         return a;
   else if ( (b>=a) && (b>=c) )
         return b;
   else
         return c;
   }
```

Attention si la signature est omise : déclaration implicite de max3 ➔ type int et erreur (alors que juste warning avec des int)

Fonction: variables locales et var. globales

- Les variables déclarées dans une fonction sont **locales** à cette fonction :
  - elles sont allouées **temporairement** dans la **pile,**
  - elles seront donc **perdues** après la fin de l'éxecution de la fonction,
    - **sauf** si la variable est précédée du mot **static** (voir slides plus loin),
  - elles ne sont pas visibles dans les fonctions appelées par cette fonction,
    - on peut donc avoir des noms de variables identiques : il n'y aura pas confusion.

- Les variables déclarées en dehors des fonctions sont **globales** :
  - elles ne sont pas allouées dans la pile,
  - toutes les fonctions peuvent y avoir accès,
  - il ne peut donc y avoir un même nom pour une var. globale et locales (nom unique),
  - à éviter le plus possible car difficile à déboguer quand le pgm devient grand.

Fonction : variables locales et var. globales

```
#include <stdio.h>


void affiche() {
   printf("in1=%f, in2=%f",in1, in2);
}

int main (void) {
   float in1=10.0,in2=20.0, res=0.0;
   affiche();
   printf("in1=%f, in2=%f",in1, in2);
    return 0;
}
```

➔erreur de compil. : in1 et in2 non décl.

var. locales ➔ var. globales

```
#include <stdio.h>
float in1=10,in2=20, res=0;

void affiche() {
   printf("in1=%f, in2=%f",in1, in2);
}

int main (void) {

   affiche();
   printf("in1=%f, in2=%f",in1, in2);
    return 0;
}
```

➔ ok car in1 et in2 sont globales donc visibles partout

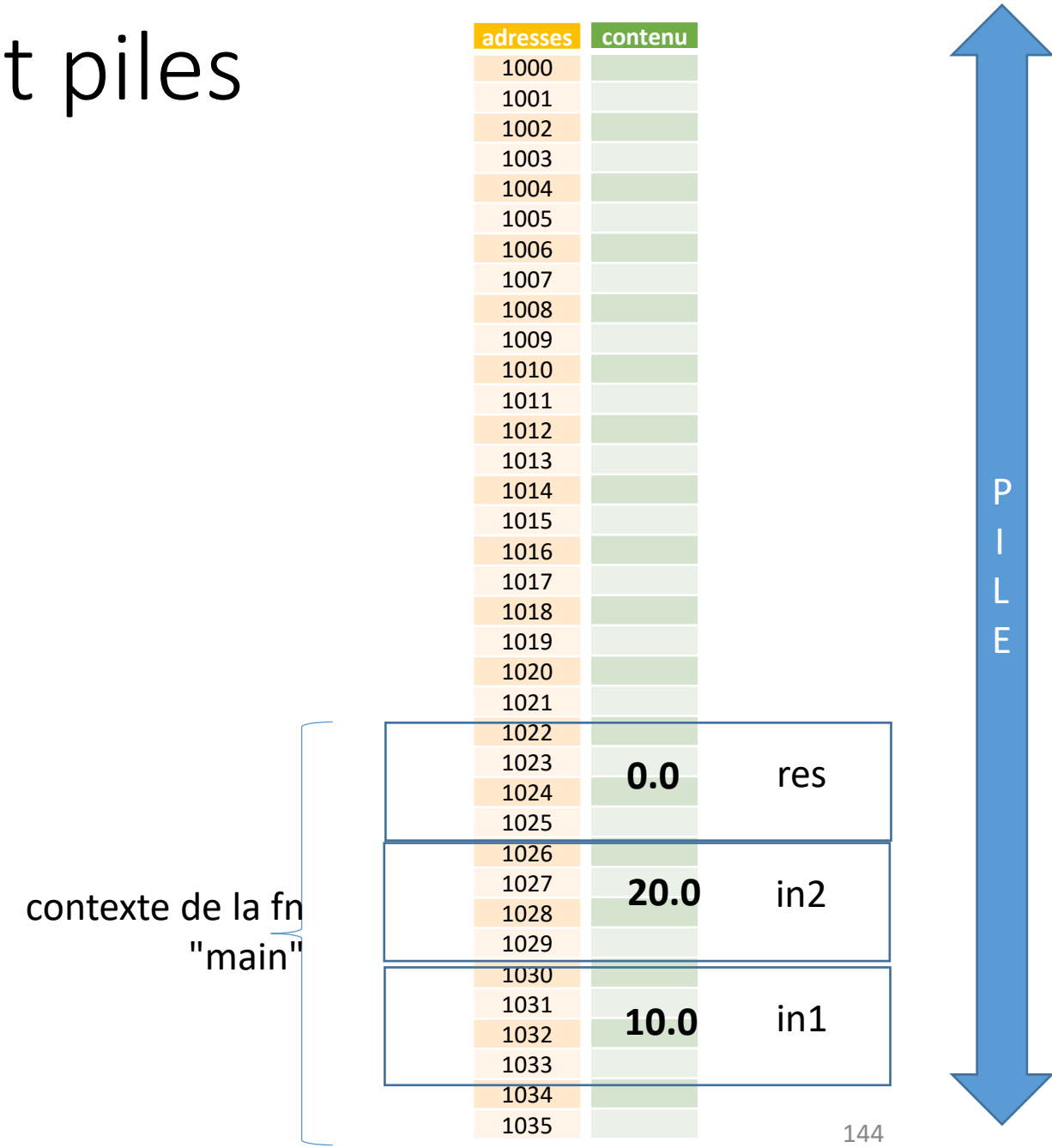
Fonction : variables locales et noms

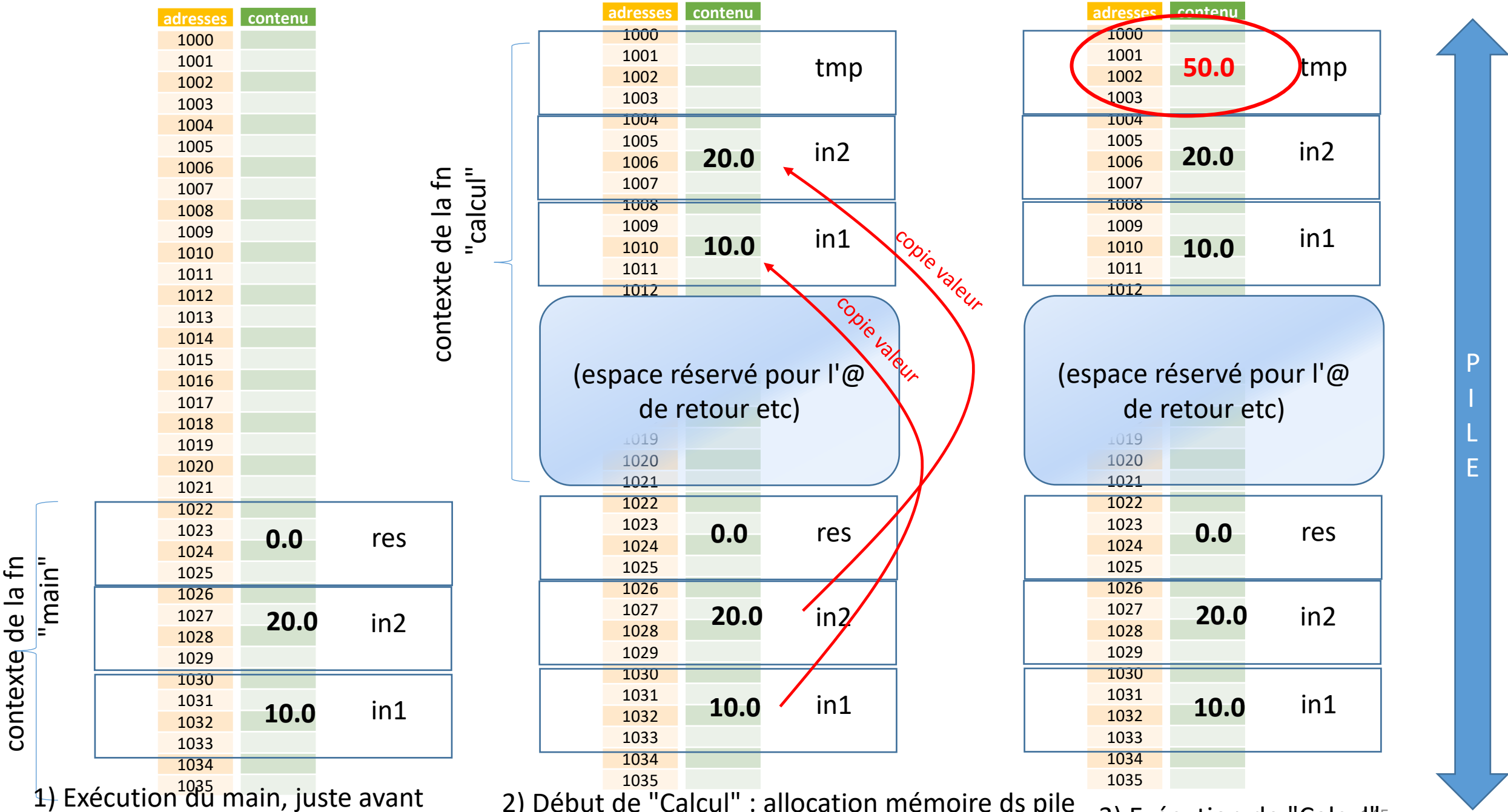
```
#include <stdio.h>
float calcul( float in1, float in2 ) {
   float tmp;
   tmp = in1/in2 *100.0;
   return tmp;
}

int main () {
   float in1=10.0,in2=20.0, res=0.0;
   res = calcul(in1, in2);
    return 0;
}
```

Les noms des variables sont identiques dans les fonctions main et calcul mais aucun problème puisque ces variables sont locales à chaque fonction.

Fonction: variables locales et piles

```
#include <stdio.h>
float calcul( float in1, float in2 ) {
   float tmp;
   tmp = in1/in2 *100.0;
   return tmp;
}
int main () {
   float in1=10.0,in2=20.0, res=0.0;
   res = calcul(in1, in2);
   return 0;
}
```

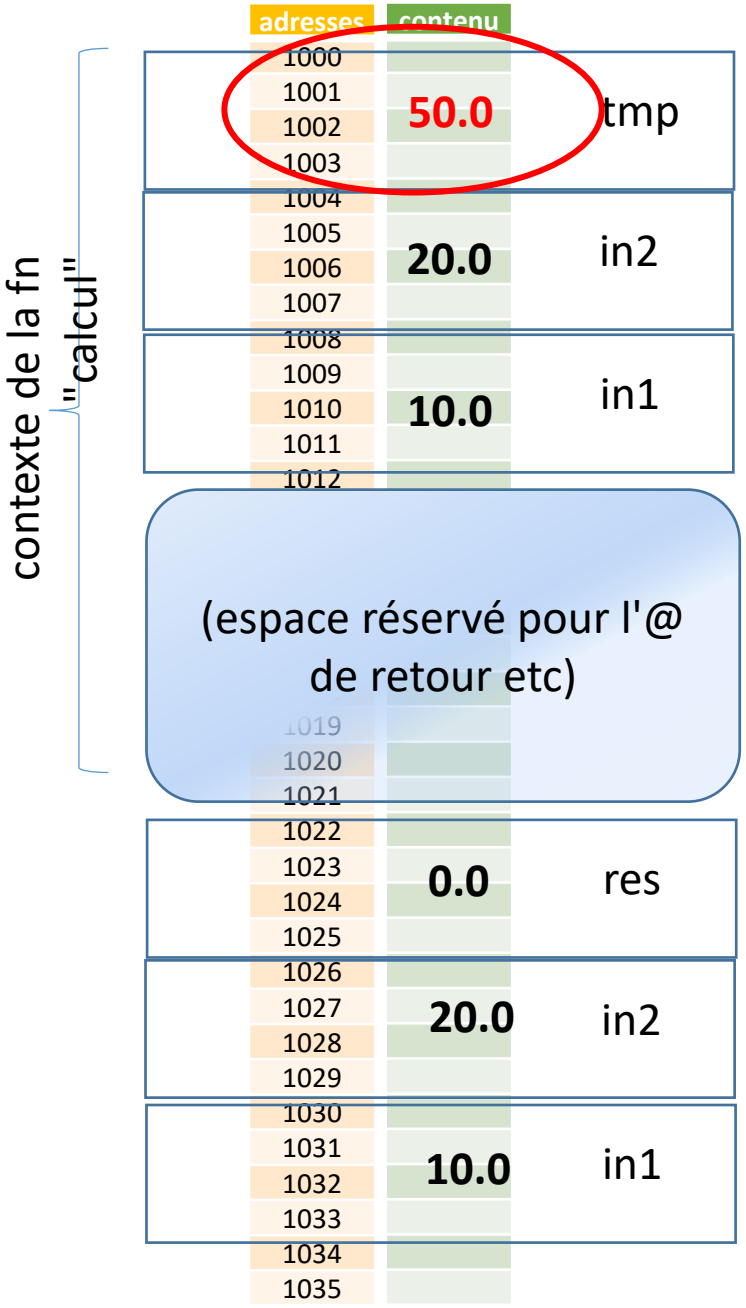




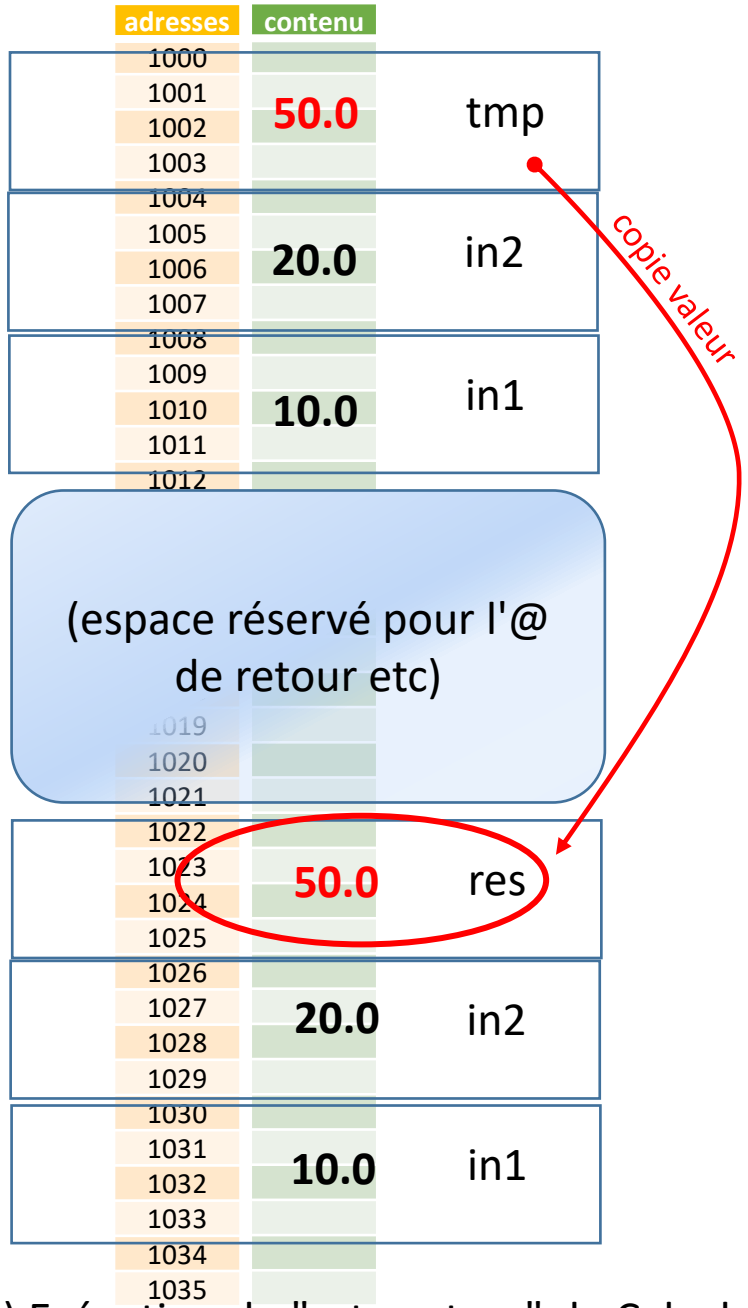
1) Exécution du main, juste avant appel à "Calcul"

2) Début de "Calcul" : allocation mémoire ds pile et recopie des valeurs de arguments en mém.

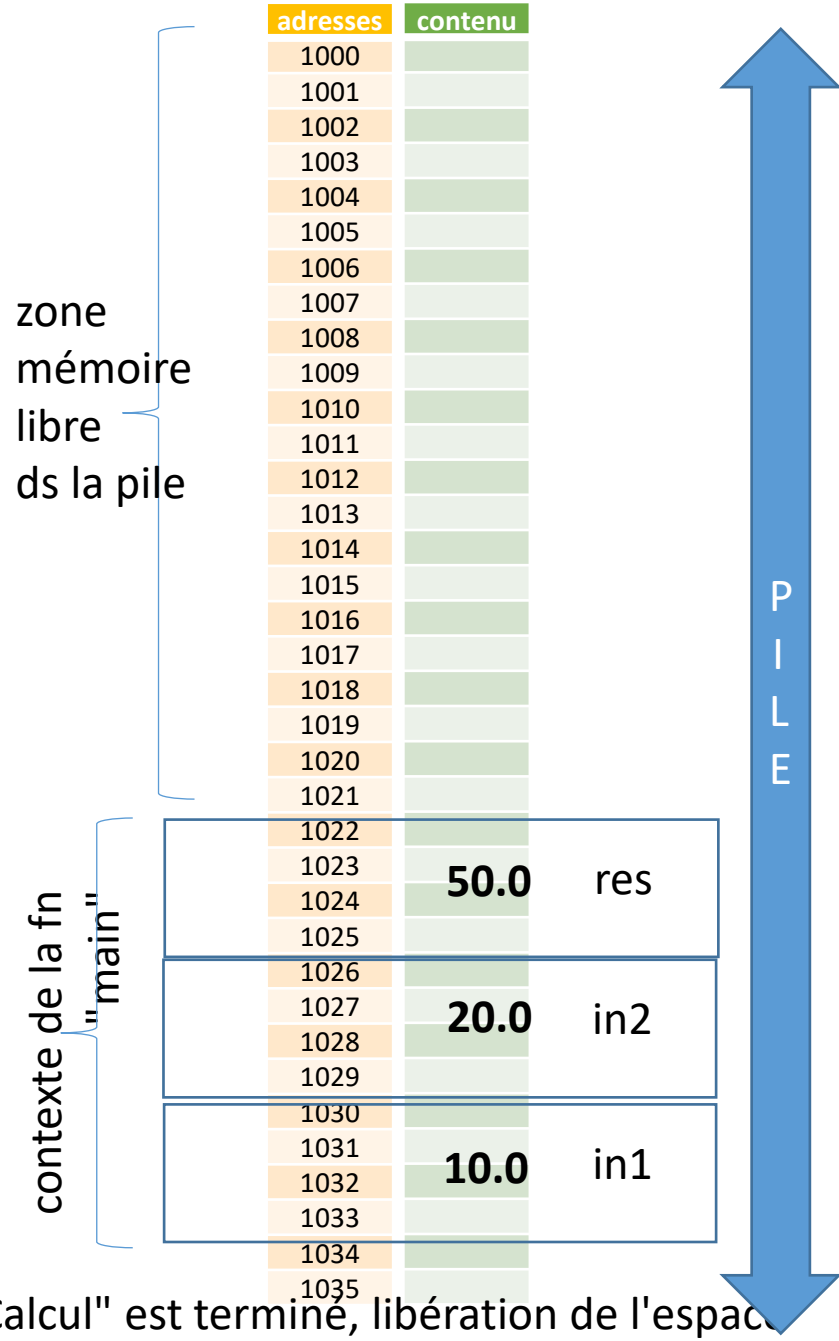
3) Exécution de "Calcul"



3) Exécution de "Calcul"
(idem slide précédent)



4) Exécution du "return tmp" de Calcul
➔ copie de la valeur de tmp ds res



5) "Calcul" est terminé, libération de l'espace sur la pile, suite des opérations du main

Fonction : exemple var. statique

```
#include <stdio.h>
void AfficheCompteur( void) {
    static int cpt=1;
    printf("cpt=%d \n",cpt);
    cpt++;
}

int main () {
    AfficheCompteur();
    AfficheCompteur();
    AfficheCompteur();
    AfficheCompteur();
    return 0;
}
```

Exécution sans le mot clef static : (à compléter)

cpt=

cpt=

cpt=

cpt=

Exécution avec cpt déclaré static : (à compléter)

cpt=

cpt=

cpt=

cpt=

Appel de fonction : assembleur

```
void Init_tab (int t[]){
      int i;
            for (i=0;i<100;i++) {
            t[i]=i;
    }
}
```

```
__semihosting_library_function:
0x00000206 4601     MOV     r1,r0
0x00000208 2000     MOVS    r0,#0x00
0x0000020A E002     B       0x00000212
0x0000020C 0082     LSLS    r2,r0,#2
0x0000020E 5088     STR     r0,[r1,r2]
0x00000210 1C40     ADDS    r0,r0,#1
0x00000212 2864     CMP     r0,#0x64
0x00000214 DBFA     BLT     0x0000020C
0x00000216 4770     BX      lr
```

```
int main() {
  int a=10,b=0;
    int tab[100];
    Init_tab (tab);
    b=a*tab[1];
    return 0;
}
```

```
main:
0x00000228 B510     PUSH    {r4,lr}
0x0000022A B0E4     SUB     sp,sp,#0x190
0x0000022C 230A     MOVS    r3,#0x0A
0x0000022E 2400     MOVS    r4,#0x00
0x00000230 4668     MOV     r0,sp
0x00000232 F7FFFFE8 BL.W    __semihosting_library_function (0x00000206)
0x00000236 9801     LDR     r0,[sp,#0x04]
0x00000238 4358     MULS    r0,r3,r0
0x0000023A 4604     MOV     r4,r0
0x0000023C 2000     MOVS    r0,#0x00
0x0000023E B064     ADD     sp,sp,#0x190
0x00000240 BD10     POP     {r4,pc}
0x00000242 0000     MOVS    r0,r0
```

Fonction récursive : exemple factoriel

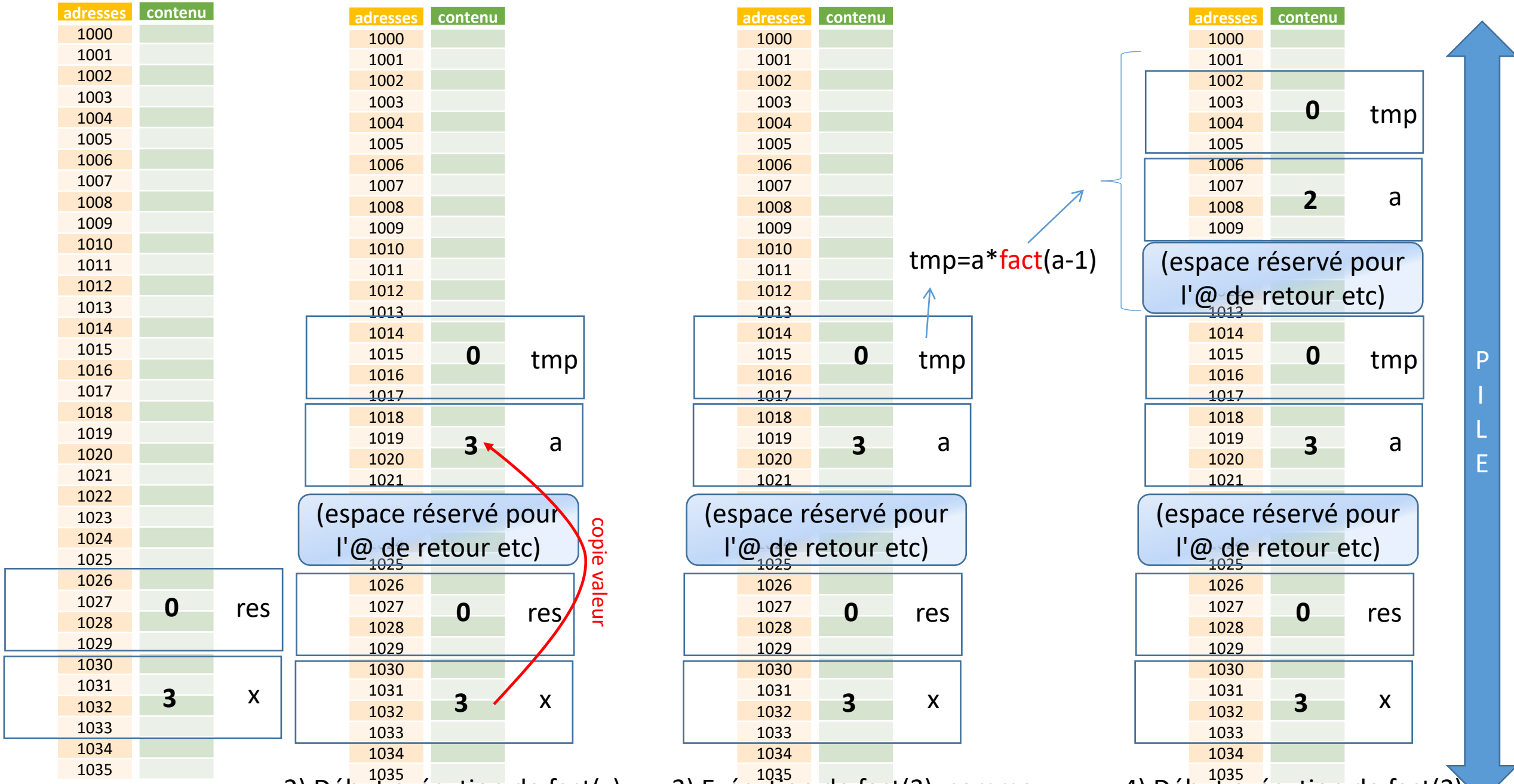
Une fonction est récursive si elle s'invoque elle même, comme ici la fonction fact qui appel fact :

```
#include <stdio.h>

int fact ( int a) {
   int tmp=0;
   if ( (a==1) || (a==0) ) {
            tmp = 1
            return tmp;
   else {
            tmp = a * fact (a-1);
            return tmp;
   }
}
```

(suite)

```
int main () {
   int x=3, res=0;
   res = fact (x);
   printf("%d!=%d",x,res);
    return 0;
}
```

3! = 6

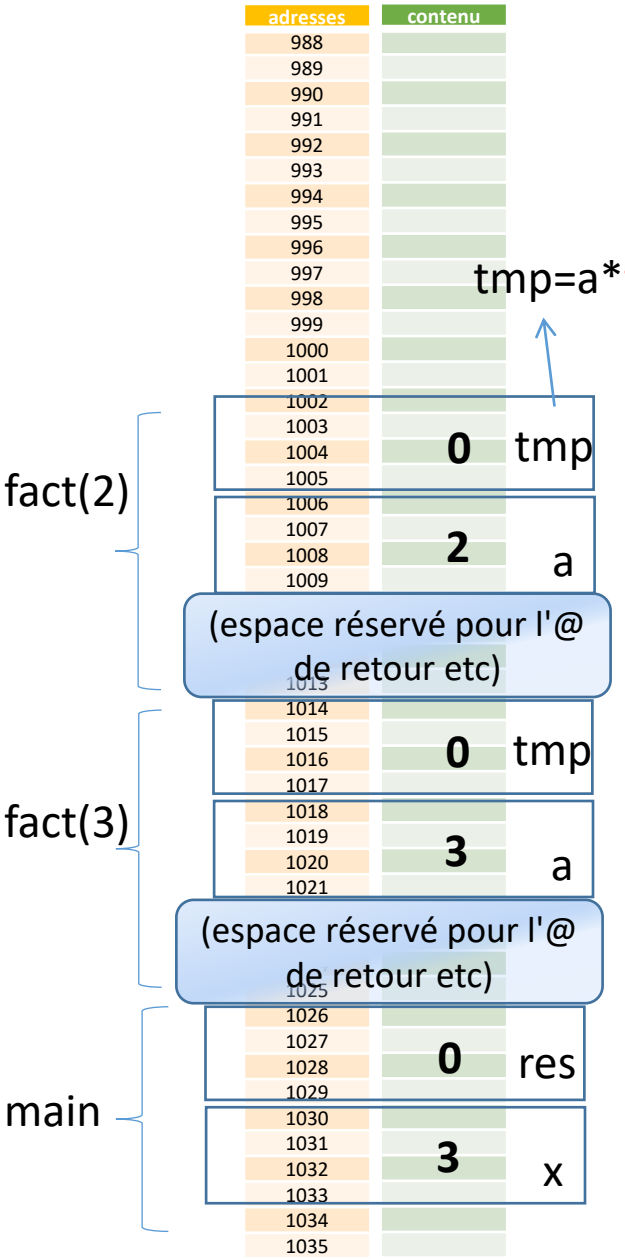


1) Exécution du main

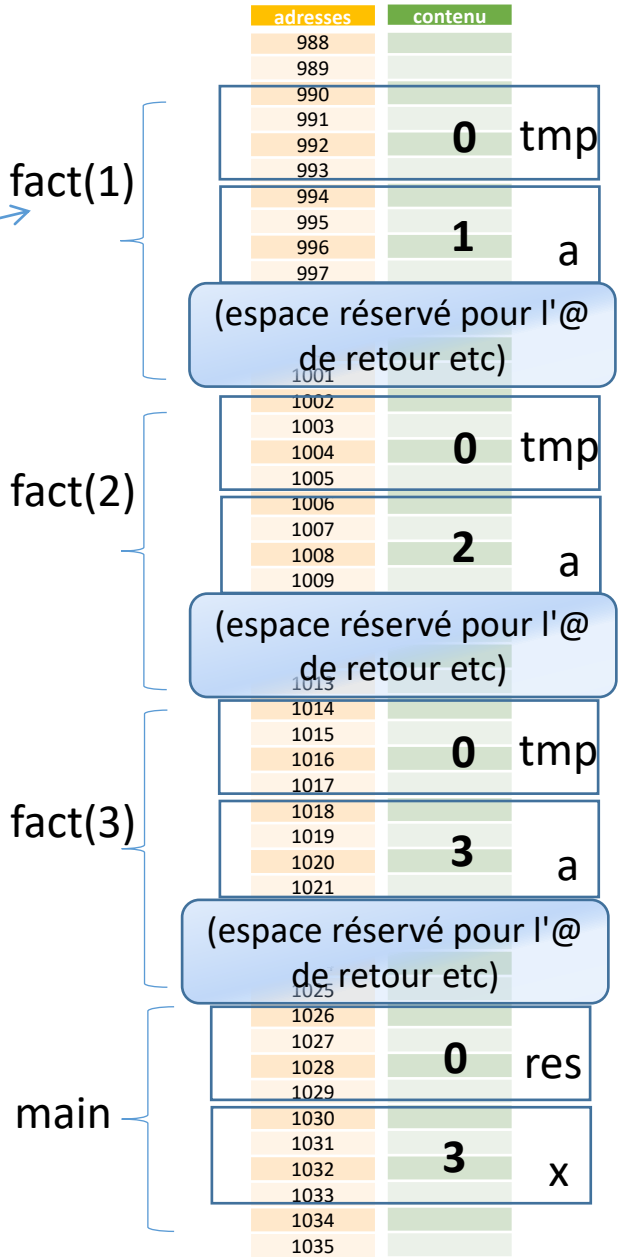
2) Début exécution de fact(x) =allocation mémoire et recopie

3) Exécution de fact(3), comme 3≠0 et 1, on doit appeler fact(3-1)

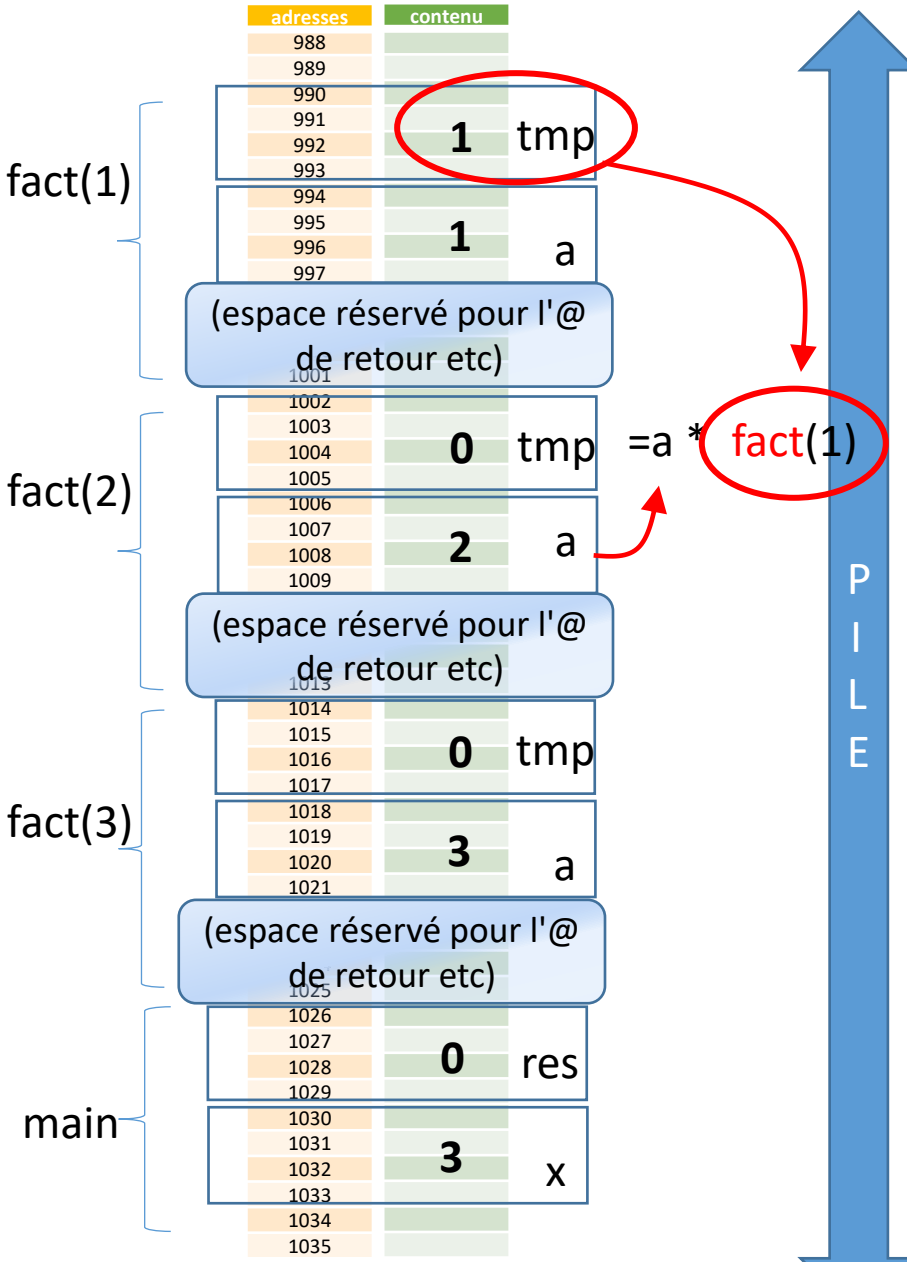
4) Début exécution de fact(2) =allocation mémoire et recopie



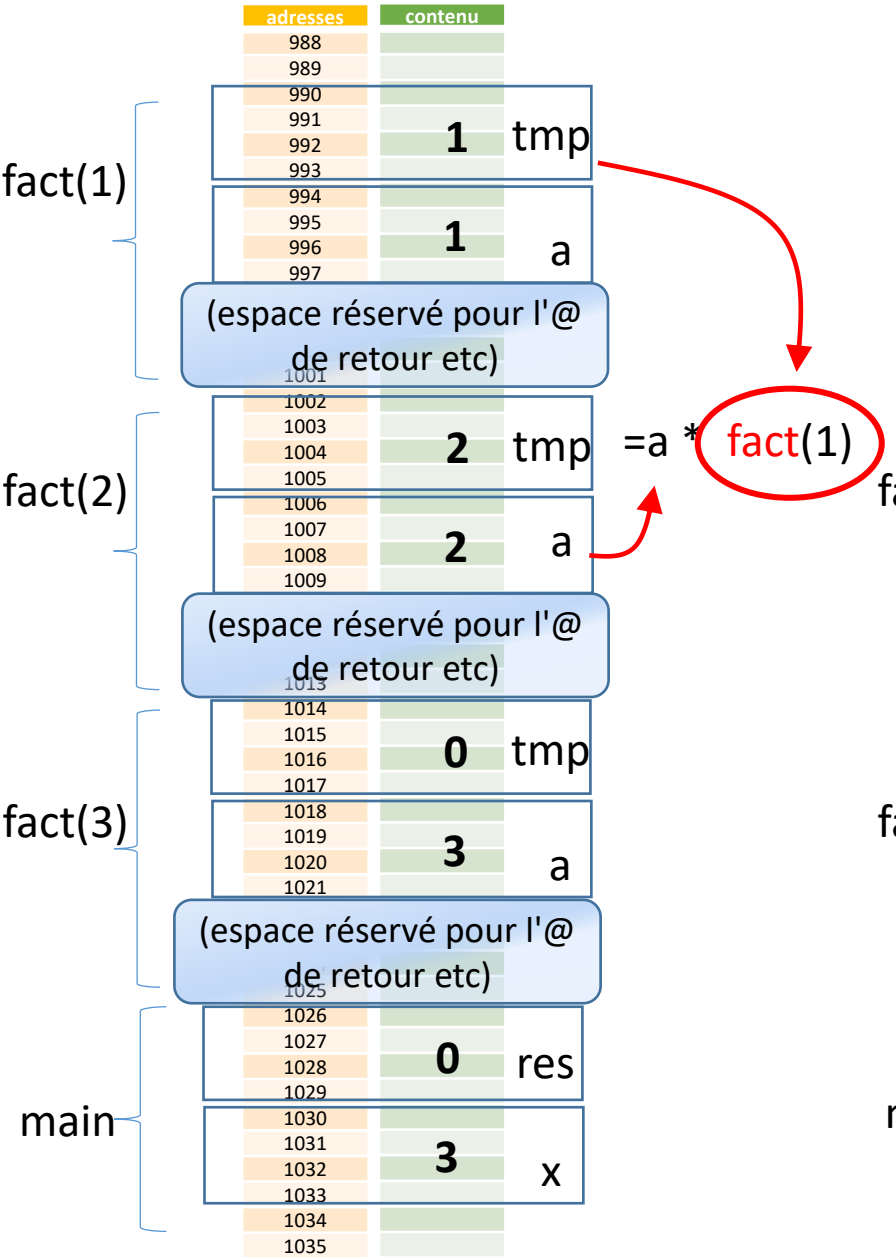
5) Exécution de fact(2), comme 2≠0 et 1, on doit appeler fact(2-1)



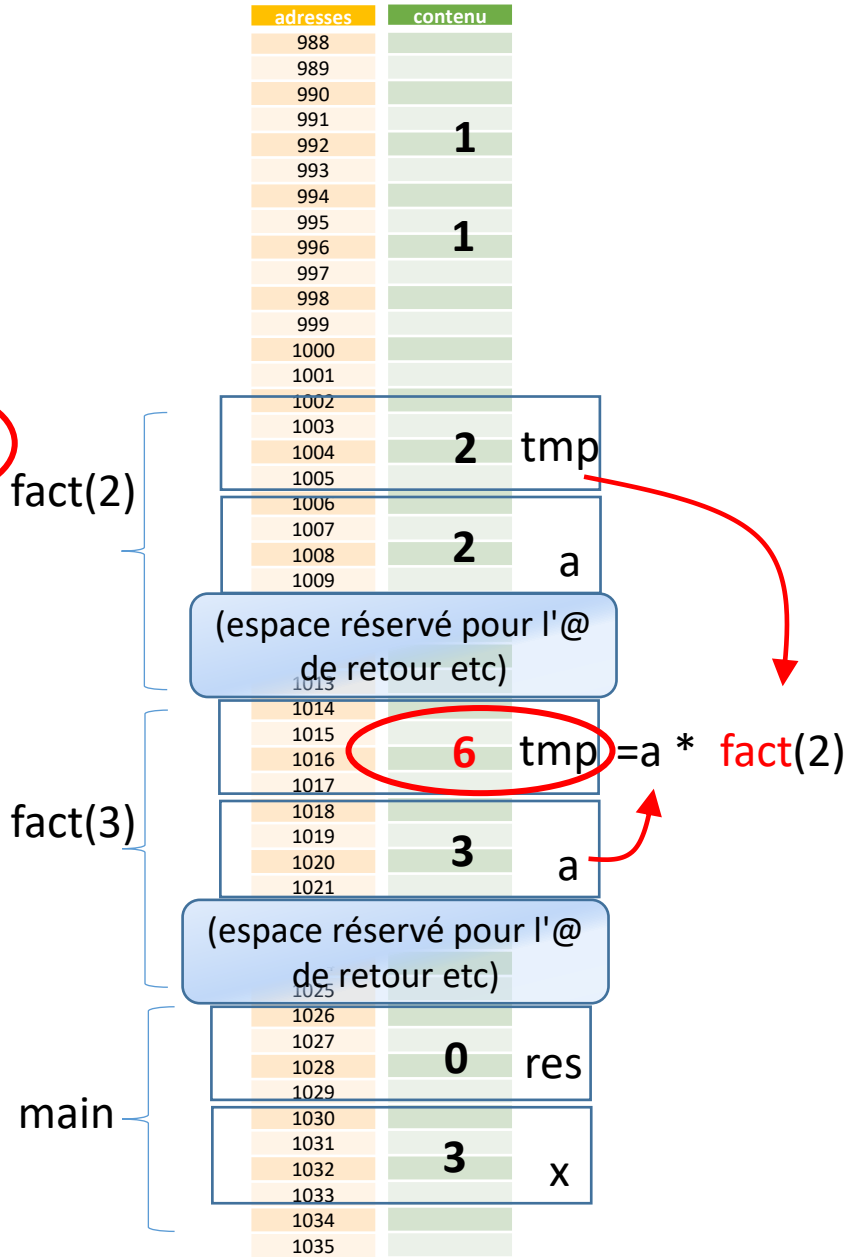
6) Début exécution de fact(1) =allocation mémoire et recopie



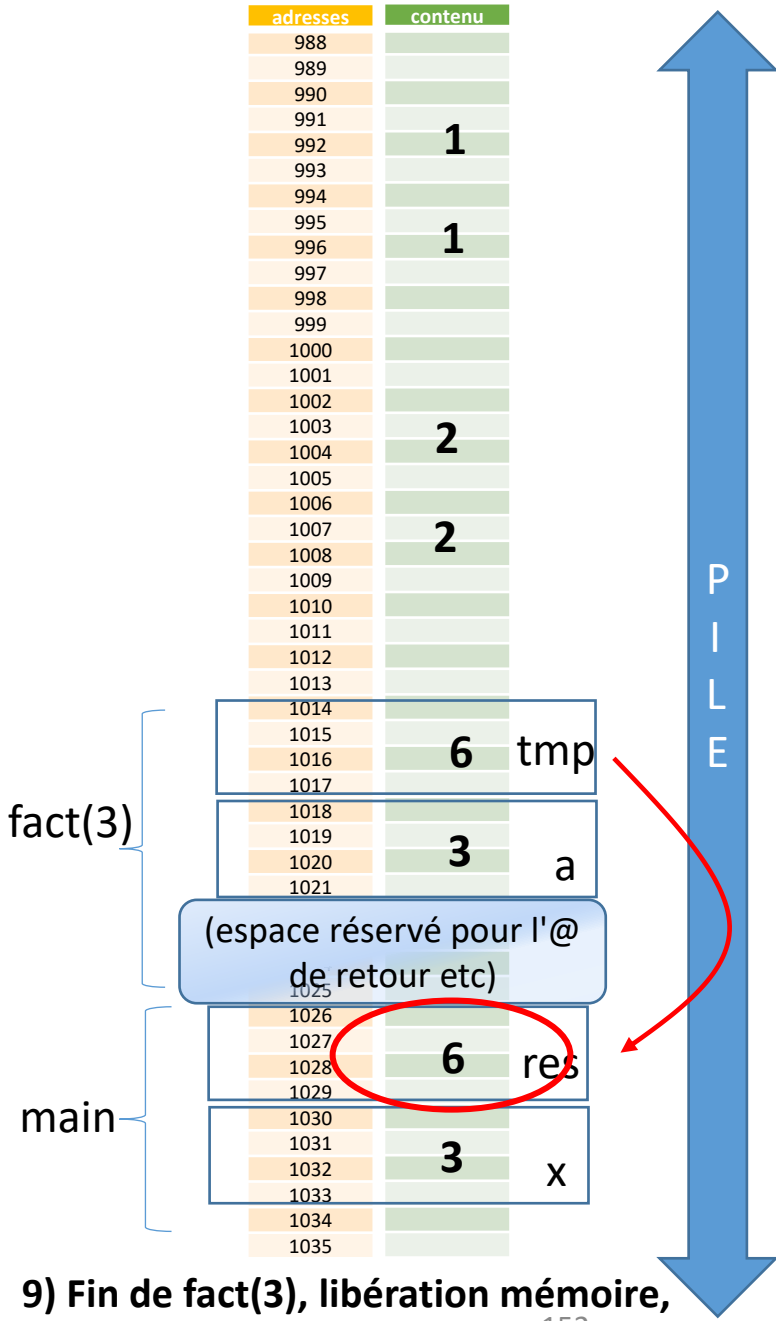
7) Fin de fact(1), if a==1 tmp=1, return tmp à fact(2), lib. mémoire



7) Fin de fact(1), if a==1 tmp=1, return tmp à fact(2), lib. mémoire

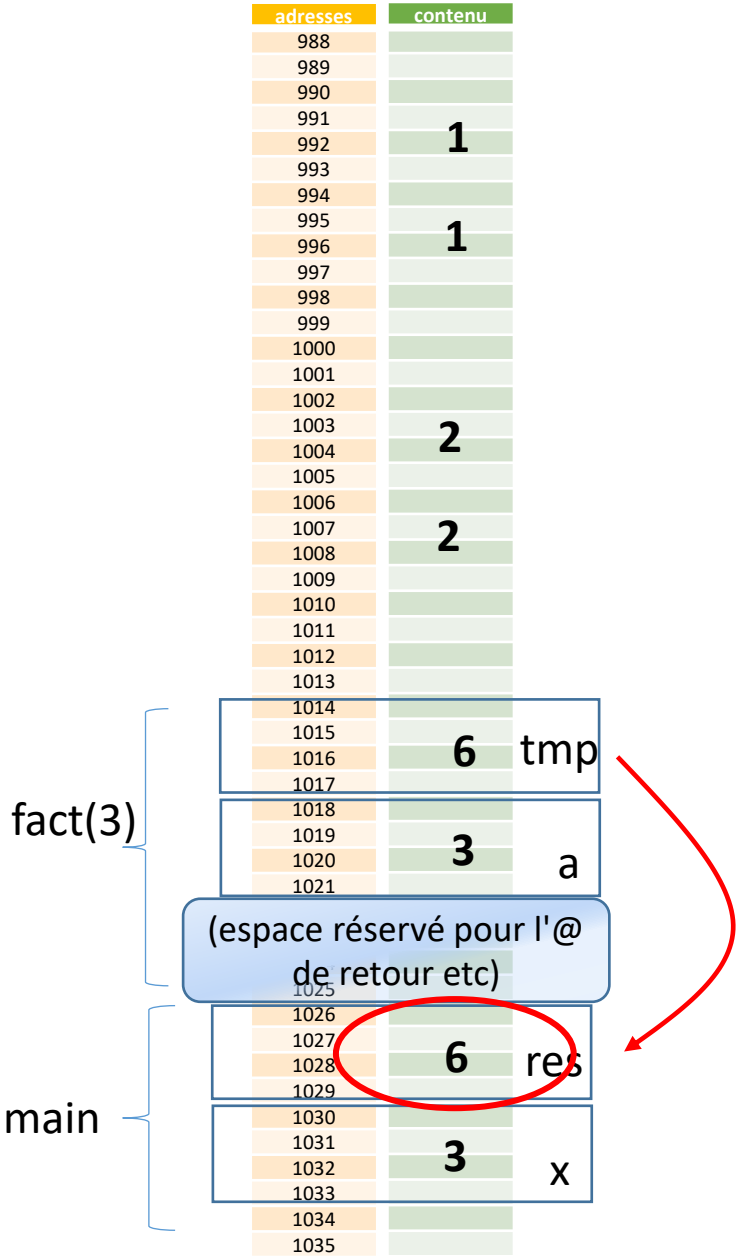


8) Fin de fact(2), libération mémoire, retour ds dans fact(3), tmp=3*2, return tmp à fact(3)

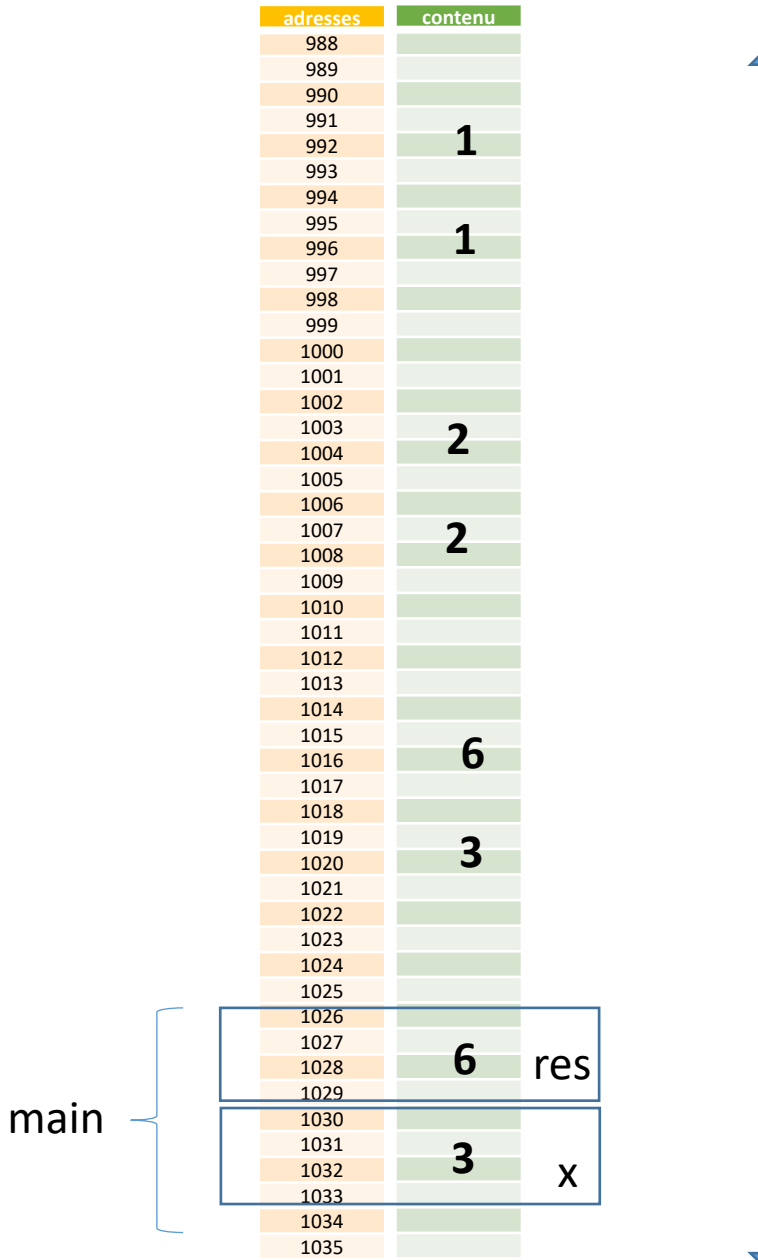


9) Fin de fact(3), libération mémoire, retour ds dans main res = tmp=6,

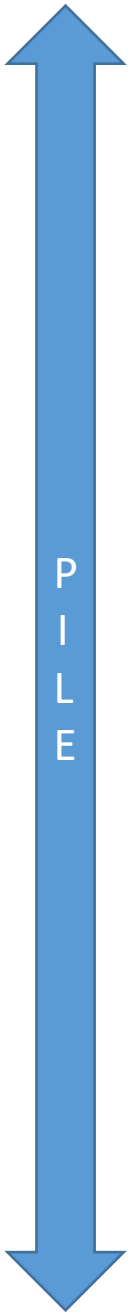
152



9) Fin de fact(3), libération mémoire, retour ds dans main res = tmp=6,



10) retour ds main, exec. suite après l l'appel à fact(x)



Sur un PC la pile peut être très grande (mécanisme de mémoire virtuelle émulée sur disque dur)

Dans l'embarqué la taille de la pile doit être défini en fonction de la mémoire physique disponible...

En fonction du pgm, difficile de connaître à priori la taille nécessaire pour la pile

Les fn récursives consomment de l'espace sur la pile

V – Pointeurs, le retour...

pointeurs et fonctions

pointeurs et tableaux

Pointeur : passage par val., la fonction permute

```
#include <stdio.h>
void permute (int a, int b) {
  int tmp;
  tmp=a;
  a=b;
  b=tmp;
}
```

```
int main () {
   int x=10, y =20;
   printf("avant : x=%d, y=%d \n", x,y);
   permute (x,y);
   printf("apres : x=%d, y=%d \n", x,y);
   return 0;
}
```

Exécution :

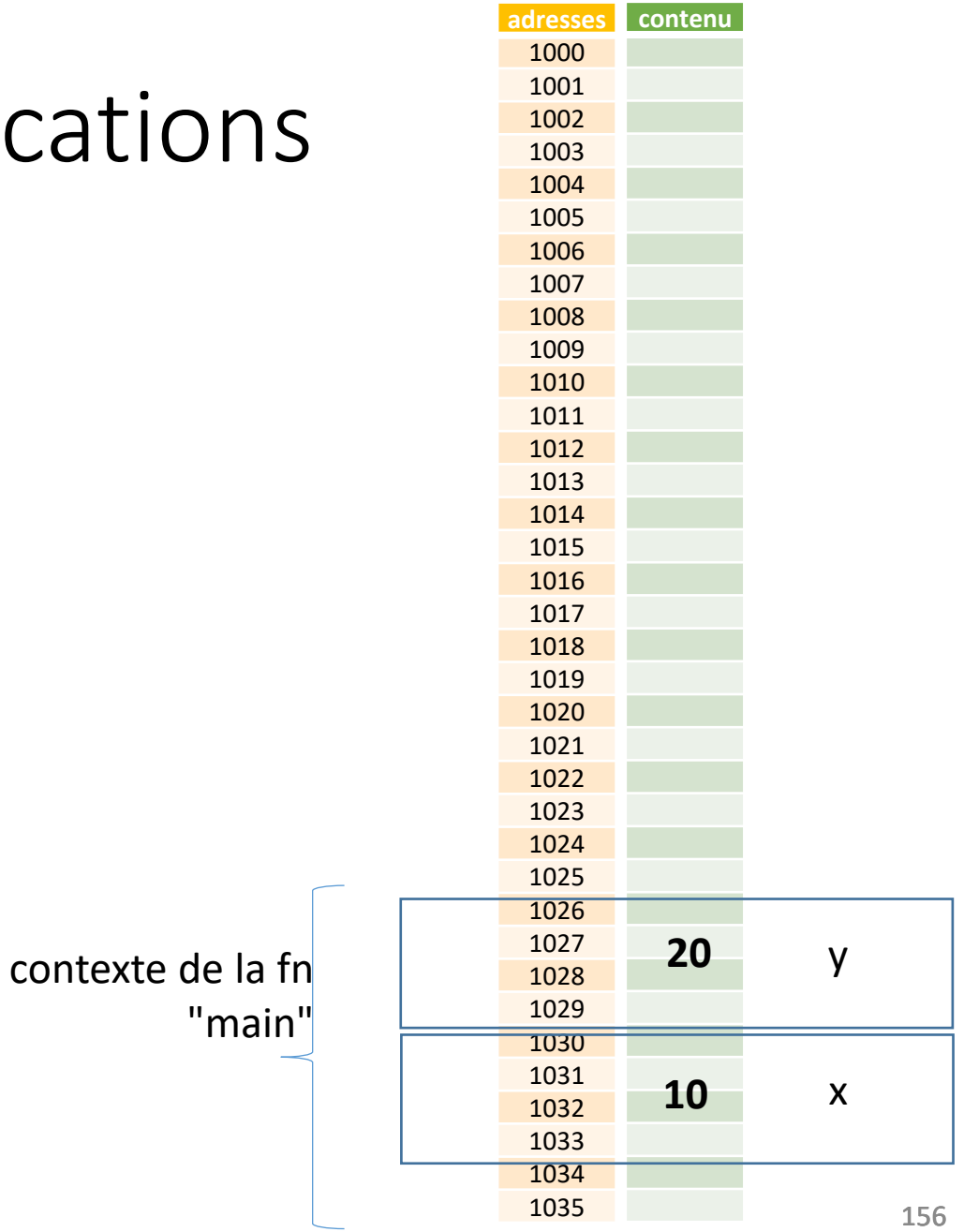
avant : x=10, y=20

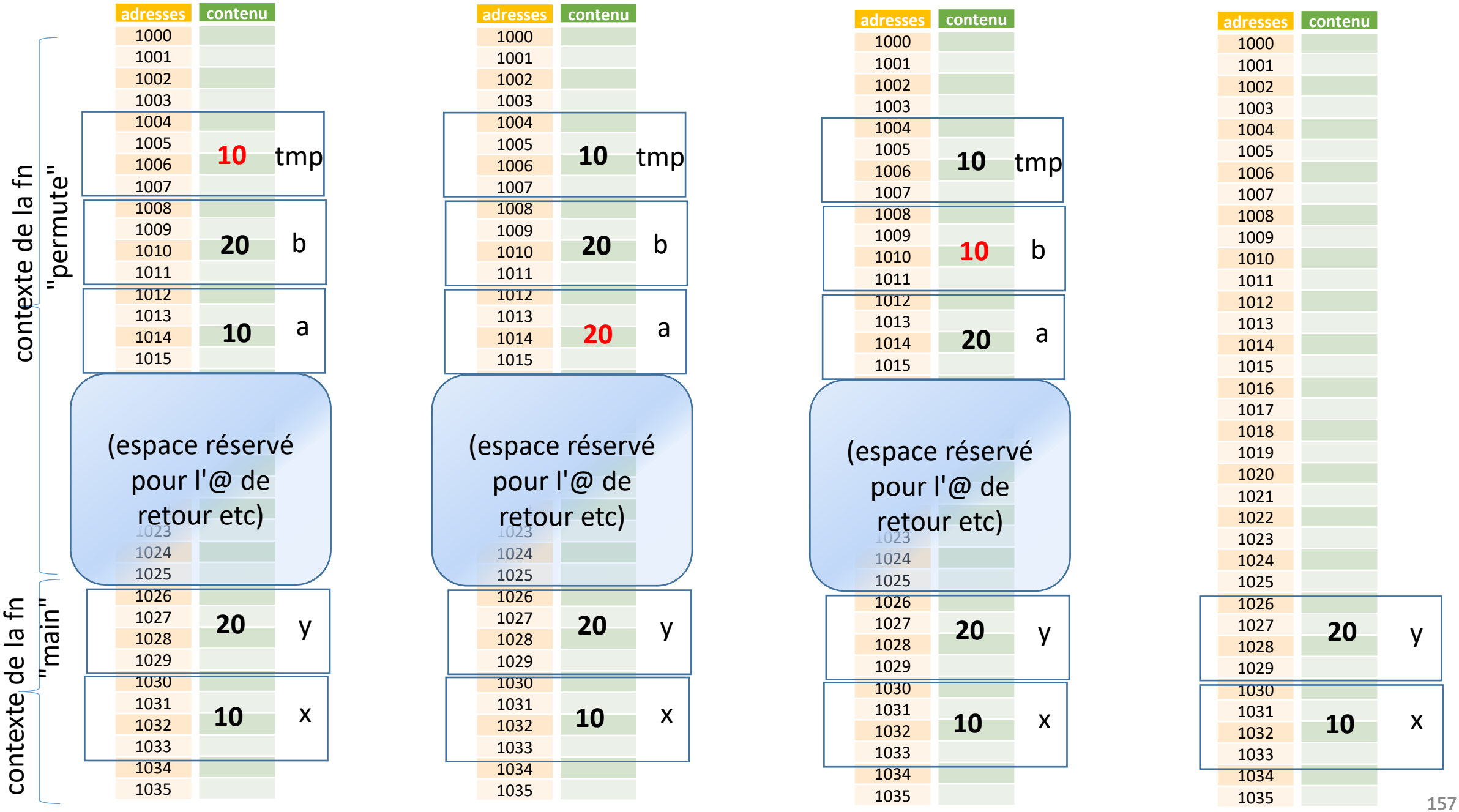
après : x=10, y=20

(alors que l'on voulais :
x=20 et y=10)

Echec de permute : explications

```
#include <stdio.h>
void permute (int a, int b) {
  int tmp;
  tmp=a;
  a=b;
  b=tmp;
}
int main () {
   int x=10, y =20;
   printf("avant : x=%d, y=%d \n", x,y);
   permute (x,y);
   printf("apres : x=%d, y=%d \n", x,y);
   return 0;
}
```




contexte de la fn "permute"
contexte de la fn "main"
adresses
contenu
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
10
tmp
20
b
10
a
(espace réservé pour l'@ de retour etc)
20
y
10
x
10
tmp
20
b
20
a
(espace réservé pour l'@ de retour etc)
20
y
10
x
10
tmp
10
b
20
a
(espace réservé pour l'@ de retour etc)
20
y
10
x
20
y
10
x
157
tmp=a
a=b
b=tmp
Fin de permute, retour au main

Pointeur : passage par @, fonction permute

(à compléter !)

```
#include <stdio.h>
void permute (int   *a, int   *b) {
  int tmp;
  tmp=  *a;
    *a =  *b;
    *b = tmp;
}
```

```
int main () {
   int x=10, y =20;
   printf("avant : x=%d, y=%d \n", x,y);
   permute (   &x,   &y);
   printf("apres : x=%d, y=%d \n", x,y);
   return 0;
}
```

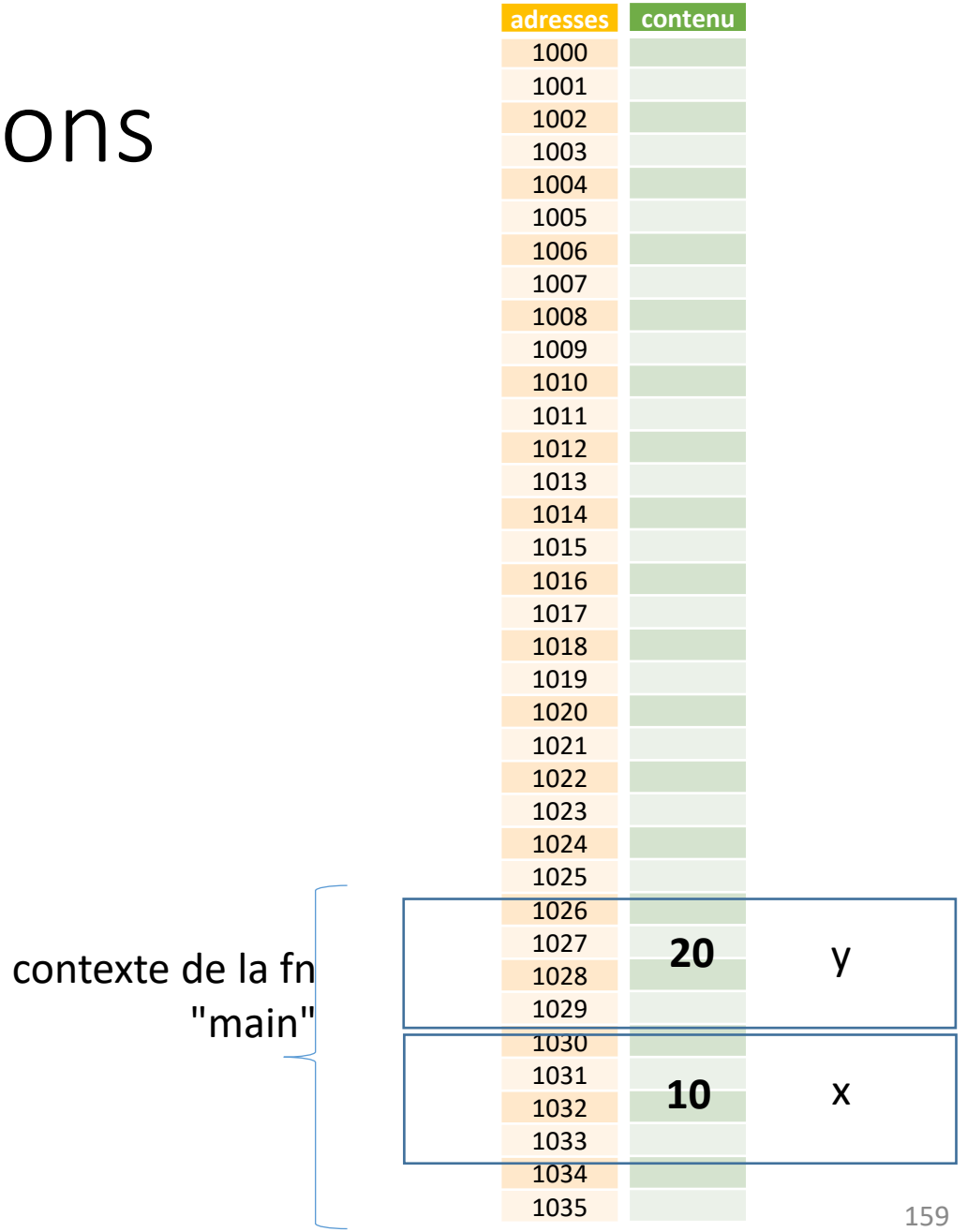
Exécution :

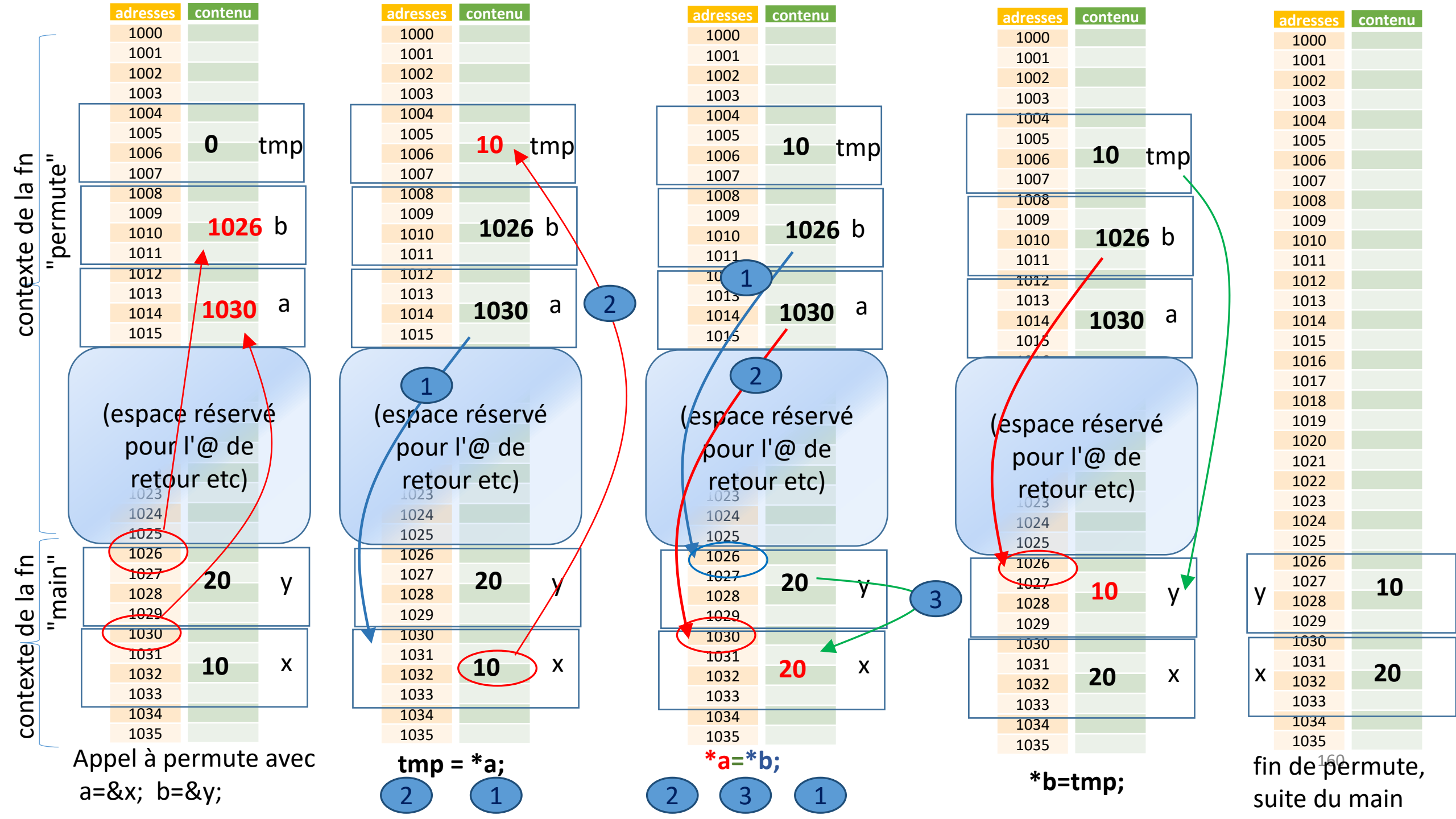
avant : x=10, y=20

apres : x=20, y=10

Passage par @ : explications

```
#include <stdio.h>
void permute (int   a, int   b) {
 int tmp;
 tmp=  a;
   a=  b;
   b=tmp;
}
int main () {
  int x=10, y =20;
  printf("avant : x=%d, y=%d \n", x,y);
  permute (   x,    y);
  printf("apres : x=%d, y=%d \n", x,y);
  return 0;
}
```




adresses
contenu
contexte de la fn "permute"
contexte de la fn "main"
0 tmp
1026 b
1030 a
(espace réservé pour l'@ de retour etc)
20 y
10 x
Appel à permute avec a=&x; b=&y;
10 tmp
tmp = *a;
*a=*b;
*b=tmp;
fin de permute, suite du main
1
2
3

Pointeurs et tableaux

Rappel :

nom du tableau = @ du 1ere élément du tableau (tab ⇔ &tab[0])

int tab[10];

printf(‘’l’adresse du tableau est %p’’, tab);

⇔printf(‘’l’adresse du tableau est %p’’, &tab[0]);

Donc tableau en argument d'une fonction = adresse du tableau

➔il n'y a donc pas de copie du tableau dans la fonction

➔la fonction accède directement au tableau passé en arg.

Pointeurs et tableaux

```
#include <stdio.h>
#define TAILLE 7

void AfficheTableau( int tab[] ) {
  int i;
  for (i=0; i<TAILLE; i++)
       printf("tab[%d]=%d \n", i, tab[i]);
}

int main () {
   int t[TAILLE]={11,22,33,44,55,66,77};
   AfficheTableau(t);
return 0;
}
```

identique

```
#include <stdio.h>
#define TAILLE 7

void AfficheTableau( int *tab ) {
    int i;
   for (i=0; i<TAILLE; i++)
       printf("tab[%d]=%d \n", i, tab[i]);
}

int main () {
   int t[TAILLE]={11,22,33,44,55,66,77};
   AfficheTableau(t);
return 0;
}
```

Mais que vaut sizeof(tab) ??

Pointeurs et tableaux : taille = danger

```
#include <stdio.h>

void AfficheTableau( int *tab ) {
    int TAILLE, i;
    TAILLE = sizeof(tab)/sizeof(int);
     printf("Taille = %d \n",TAILLE);
    for (i=0; i<TAILLE; i++)
        printf("tab[%d]=%d \n", i, tab[i]);
}

int main () {
    int t[7]={11,22,33,44,55,66,77};
   AfficheTableau(t);
return 0;
}
```

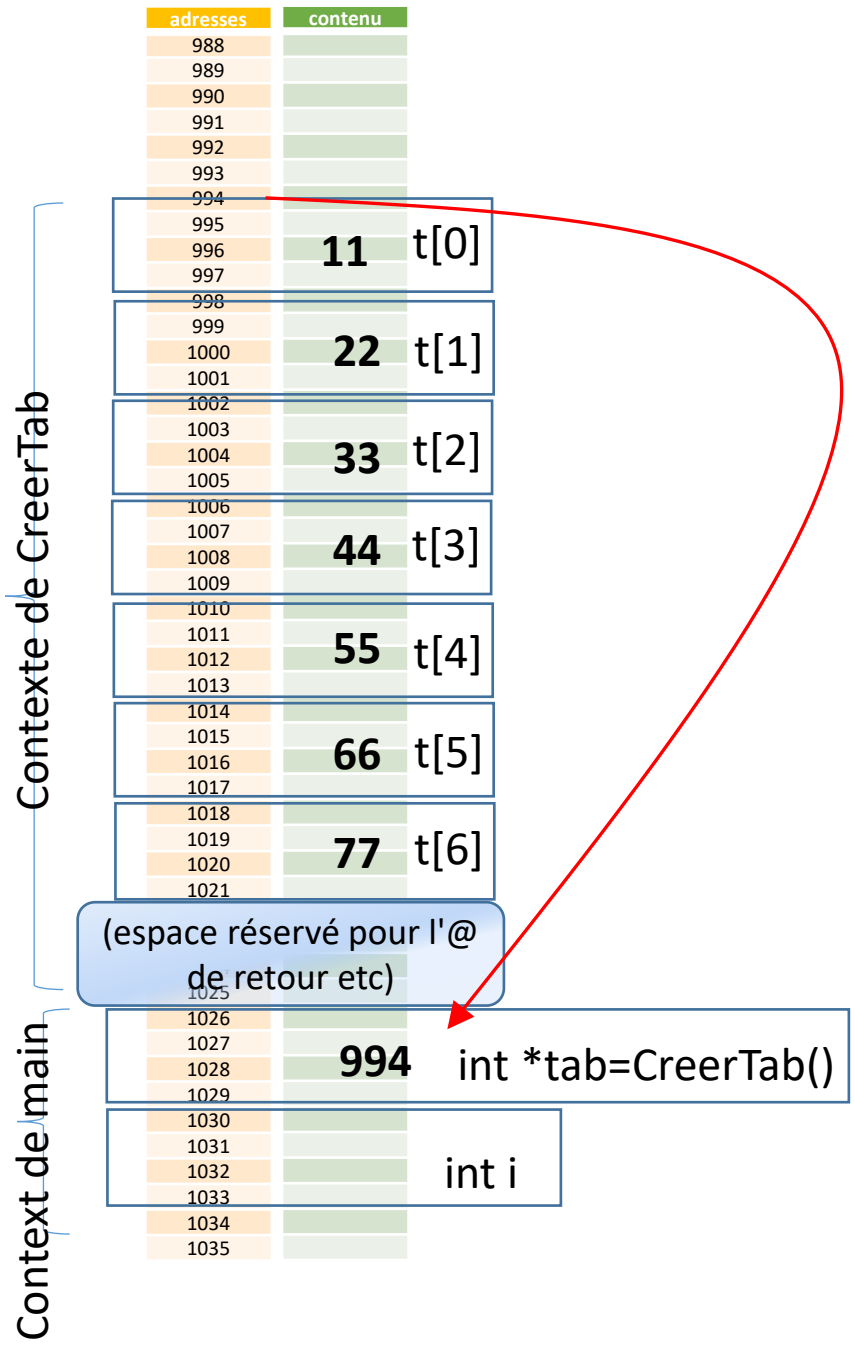
```
#include <stdio.h>

void AfficheTableau( int *tab, int TAILLE ) {
   int i;
   printf(" TAILLE = %d \n",TAILLE);
   for (i=0; i<TAILLE; i++)
        printf("tab[%d]=%d \n", i, tab[i]);
}

int main () {
    int t[7]={11,22,33,44,55,66,77};
   AfficheTableau(t, sizeof(t)/sizeof(int));
return 0;
}
```

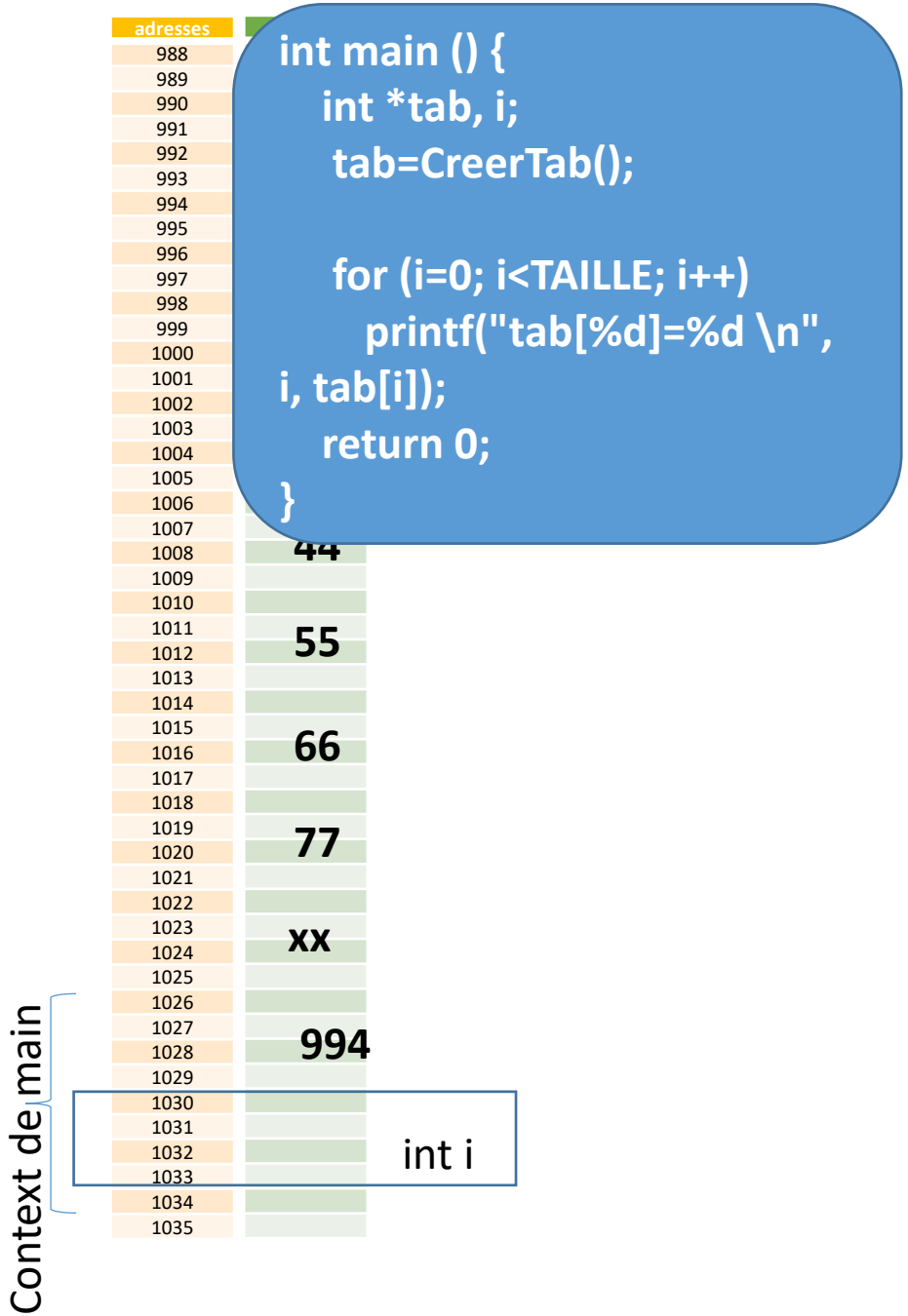
Il faut passer la taille du tableau en argument, ou en variable globale, ou utiliser #define

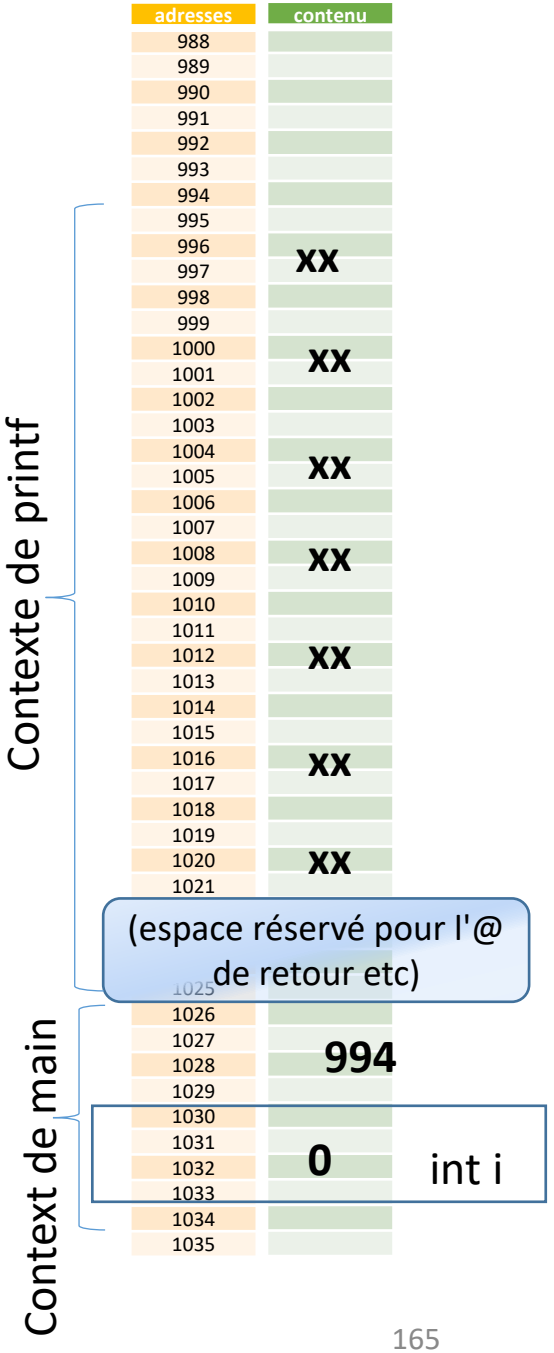
Pointeurs et tableaux : tableau local = danger

```
#include <stdio.h>
#define TAILLE 7

int* CreerTab() {
      int t[TAILLE]={11,22,33,44,55,66,77};
      return t; //⇔ return &t[0];
}

int main () {
    int *tab, i; ⇔ int tab[ ];
     tab=CreerTab();
    for (i=0; i<TAILLE; i++)
       printf("tab[%d]=%d \n", i, tab[i]);
   return 0;
}
```

Cet exemple est bogué : en effet où en mémoire le tableau t est-il alloué ?


adresses
contenu
Contexte de CreerTab
11
t[0]
22
t[1]
33
t[2]
44
t[3]
55
t[4]
66
t[5]
77
t[6]
(espace réservé pour l'@ de retour etc)
Context de main
994
int *tab=CreerTab()
int i


adresses
int main () {
int *tab, i;
tab=CreerTab();
for (i=0; i<TAILLE; i++)
printf("tab[%d]=%d \n",
i, tab[i]);
return 0;
}
11
22
33
44
55
66
77
xx
994
Context de main
int i


adresses
contenu
Contexte de printf
xx
xx
xx
xx
xx
xx
(espace réservé pour l'@ de retour etc)
Context de main
994
0
int i

Pointeurs et tableaux : tableau local = danger

Mais alors comment une fonction peut elle remplir un tableau :

il faut lui donner l'adresse du tableau

Exemple : fonction scanf

Cette fonction permet de saisir une chaîne de caractère ou un nombre au clavier et de le stocker à l'adresse qui a été donnée en argument

Pointeurs : fonction scanf

Dans le terminal, taper **man scanf**

SCANF(3) Manuel du programmeur Linux SCANF(3)

NOM

scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf - Entrées formatées.

SYNOPSIS

#include <stdio.h>

int scanf (const char *format, ...);

...

DESCRIPTION

Les fonctions de la famille **scanf** analysent leurs entrées conformément au *format* décrit plus bas...../...

VALEUR RENVOYÉE

Ces fonctions renvoient le nombre d'éléments d'entrées correctement assignés. ../..

Scanf : formats

- Comme précédemment : %d entier, %u non signé
- Mais aussi (depuis C99) :

  %h pour un short
  %hu pour un short non signé
  %hh pour un char
  %hhu pour un char non signé

Pointeurs : fonction scanf

Exemple pour saisir un entier :

```
#include <stdio.h>
int main () {
   int  a, res;
   printf("Entrer  la valeur de a : ");
   res=scanf("%d", &a);
   printf("a=%d \n", a);
   printf("res=%d",res);
   return 0;
}
```

Exemple pour saisir une chaîne de caractères :

```
#include <stdio.h>
int main () {
  int res;
  char Mot[30];
  printf("Entrer  un mot : ");
  res=scanf("%s", Mot);
  printf("Mot = %s \n", Mot);
  printf("res=%d",res);
  return 0;
}
```

L'argument de scanf doit être une adresse pour que scanf puisse modifier l'argument

ATTENTION : RES NE CONTIENT PAS LA VALEUR SAISIE AU CLAVIER !!

Pointeurs : fonction scanf

Exemple pour saisir les valeurs d'un tableau

```
#include <stdio.h>
int main () {
  int i, tab[10];
  for (i=0; i<10; i++) {
     printf("Entrer  le nombre tab[%d] : ");
     scanf("%d", &tab[i]);
  }
../..
return 0;
}
```

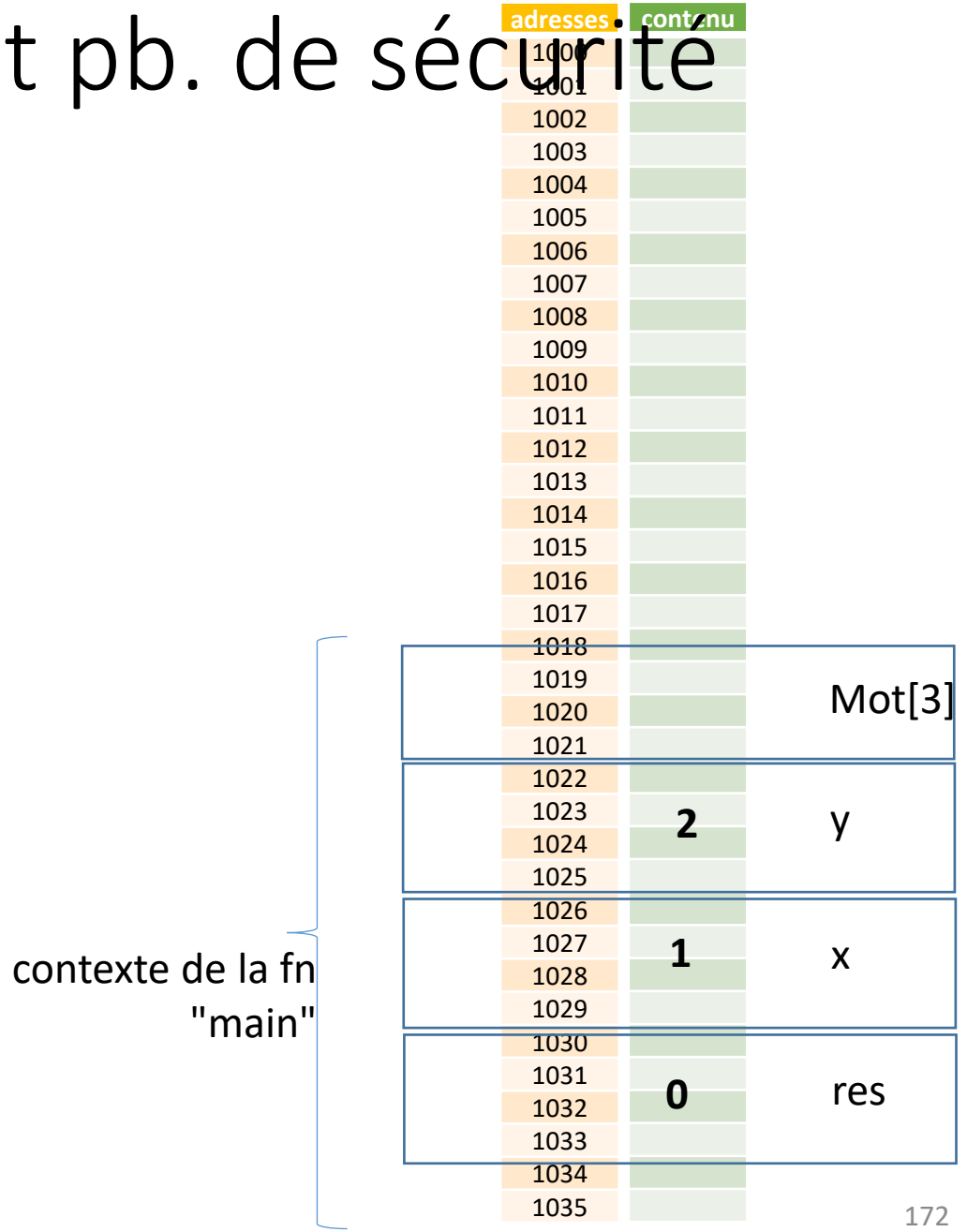
Pointeurs : fonction scanf et pb. de sécurité

Que ce passe t-il si la chaine de caractère saisie dépasse la taille du tableau alloué pour stocker cette chaîne ?

➔dépassement de tableau (buffer overflow) = risque de plantage, bug, "segmentation fault", piratage...

Pointeurs : fonction scanf et pb. de sécurité

```
#include <stdio.h>
int main () {
  int res=0,x=1,y=2;
  char Mot[3];

  printf("Entrer  un mot : ");
  res=scanf("%s", Mot);

  printf("Mot = %s \n", Mot);
  printf("res=%d \n",res);
  printf("res=%d \n",x);
  printf("res=%d \n",y);
  return 0;
}
```



Pointeurs : fonction scanf et pb. de sécurité

Il faut contrôler le nombre de caractères saisis dans le scanf

1ère solution : imposer le nombre de caractères à lire avec

scanf("%2s", Mot); 2 est placé en % et s : il indique le nb de caract

Si on tape plus de 2 caractères, ils sont laissés dans le buffer système....et seront considérés comme les caractères saisis lors du prochain scanf ☹

➔ il faut donc les lire sans les utiliser :

```
char c;
while ((c = getchar ()) != '\n' && c != EOF);
```

Pointeurs : fonction scanf et pb. de sécurité

```
#include <stdio.h>
int main () {
  int res=0;
  char Mot[4], c;

  printf("\nEntrer  un mot : ");
  res=scanf("%3s", Mot);
  while ((c = getchar ()) != '\n' && c != EOF) ;
  printf("Mot = %s res=%d\n", Mot,res);

  printf("\nEntrer  un mot : ");
  res=scanf("%3s", Mot);
  while ((c = getchar ()) != '\n' && c != EOF) ;
  printf("Mot = %s res=%d\n", Mot,res);

return 0;
}
```

Pointeurs : fonction scanf et pb. de sécurité

2) Remplacer scanf par fgets quand il s'agit des chaines de caract. :

char *fgets (char *s, int *size*, FILE *stream);

fgets() lit au plus *size - 1* caractères depuis *stream* et les place dans le tampon pointé par *s*. La lecture s'arrête après **EOF** ou un retour-chariot. Si un retour-chariot (newline) est lu, il est placé dans le tampon. Un caractère nul « **\0** » est placé à la fin de la ligne.

VALEUR RENVOYÉE

fgetc(), **getc**() et **getchar**() renvoient un caractère, lu comme un **unsigned char**, et transformé en **int**, ou **EOF** à la fin du fichier, ou en cas d'erreur.

Passage de tableaux à 2D

Tableaux 1D : tab[i] ⇔ *(tab+i)

Rappels :

Tableaux 2D :

Déclaration : **type NomTab[Nbligne][NbColonnes]; /*les elt d'une ligne sont contigues en mem*/**

&tab[i][j] = &tab[0][0] + i* **NbColonnes** +j

tab[i][j] ⇔ *(tab + i* **NbColonnes** + j)

Le compilateur a besoin de la taille d'une ligne (= **NbColonnes**) pour trouver l'élément !

Quand on passe un tableau 2D en argument à une fonction

➔ la fonction doit donc connaitre la taille de la ligne du tableau 2D

Passage de tableaux à 2D

```
#include <stdio.h>
void AfficheTableau( int tab[][3] , int l , int c) {c)
  int i,j;
  for (i=0; i<l; i++)
    for (j=0;j<c;j++)
       printf("tab[%d][%d]=%d \n", i,j, tab[i][j]);
}

int main () {
    int t[2][3] ={ {11,12,13}, {21,22,23}};
    AfficheTableau(t,2,3);
    return 0;
}
```

Notez Bien :

on ne peut faire void AfficheTableau(int tab[][] , int l , int c)

Au moment de la compilation,
il faut passer la taille de la ligne (=nb de colonnes)

Inconvénient : il n'est pas toujours connu…, possibilité = passer par un pointeur

Passage de tableaux à 2D

```
#include <stdio.h>
void AfficheTableau( int *tab , int l , int c) {
  int i,j;
  for (i=0; i<l; i++)
    for (j=0;j<c;j++)
       printf("tab[%d][%d]=%d \n", i,j, *(tab+i*c+j)  );
}

int main () {
    int t[2][3] ={ {11,12,13}, {21,22,23}};
    AfficheTableau(t,2,3);
    return 0;
}
```

Pointeurs : arithmétique des pointeurs

int t[5]={11,22,33,44,55};

t contient l'adresse de tab[0]

donc *t ⇔ tab[0] ⇔ 11

Et pour tab[1] ?

➔ quelle est l'adresse de tab[1] ?

cad quelle est &tab[1] ?

*(t+1) ? ~~ou *(t+4)~~ ? (sachant que la taille d'un int est 4 octets)

Pointeurs et tableaux : accès, arithmétique des pointeurs

```
#include <stdio.h>
int main () {
   int i=0, t[5]={11,22,33,44,55};
   printf("t[%d]=%d \n", i, *(t+i) );
    i++;
   printf("t[%d]=%d \n", i, *(t+i) );
    i++;
    printf("t[%d]=%d \n", i, *(t+i) );
    i++;
    return 0;
}
```

```
#include <stdio.h>
int main () {
  int i=0, t[5]={11,22,33,44,55};
  printf("&t[%d] =%p \n", i, &t[0] );
  printf("t+%d = %p \n",i,  t+i);
  i++;
  printf("t+%d = %p \n", i, t+i);
  i++;
  printf("t+%d = %p \n", i, t+i);
  i++;
 return 0;
}
```

&t[0] =0xbfdac5bc t+0 = 0xbfdac5bc t+1 = 0xbfdac5c0 t+2 = 0xbfdac5c4

Pointeurs : arithmétique des pointeurs

L'addition prend en compte le type du pointeur (le type pointé) :
le compilateur remplace l'addition par "addition * sizeof(type pointé)"

```
#include <stdio.h>
int main () {
  char *p=1;
  printf("p=%p \n", p);
  p++;
  printf("p=%p \n", p);
  return 0;
}
```

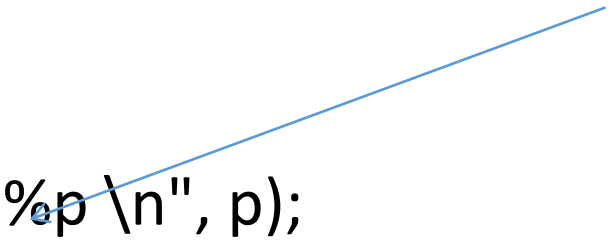
```
#include <stdio.h>
int main () {
  short*p=1;
  printf("p=%p \n", p);
  p++;
  printf("p=%p \n", p);
  return 0;
}
```

```
#include <stdio.h>
int main () {
  int *p=1;
  printf("p=%p \n", p);
  (char *) p++;
  printf("p=%p \n", p);
  return 0;
}
```

Pointeurs : arithmétique des pointeurs

Comment contrôler (forcer) l'arithmétique des pointeurs ?

```
#include <stdio.h>
int main () {
  int *p;
  p=1;
  printf("p=%p \n", p);
  p = (char*) p + 1 ;
  printf("p=%p \n", p);
  return 0;
}
```



CAST : on indique au compilateur de traiter ce qui suit comme un pointeur vers un char (un char *)

alors que c'est un pointeur vers un int

Pointeurs et structures

Il est possible d'accéder directement aux champs d'une structure à partir d'un pointeur vers une structure : " pointeur **->** champs "

Exemple :

```
#include<stdio.h>
struct Pixel {
   int x;
   int y;
};
int main() {
   struct Pixel P1;
   struct Pixel *pointeur;
   P1.x = 10;
   P1.y=20;
   pointeur = &P1;

   /*acces par pointeur : */
   printf("champs x= %d \n", (*pointeur).x);
   printf("champs y =%d \n", pointeur ->y);
   return 0;
}
```

Tableau de pointeurs

int tab[10]; /* tableau de 10 int */

int *tab[10]; /* tableau de 10 pointeurs sur des int */

Rappel : passage d'un tableau d'int en argument

Ou int tab[ ]

```
#include <stdio.h>
void AfficheTableau( int *tab, int TAILLE ) {
  int i;
  for (i=0; i<TAILLE; i++)
      printf("tab[%d]=%d \n", i, tab[i]);
}

int main () {
   int t[7]={11,22,33,44,55,66,77};
   AfficheTableau(t, sizeof(t)/sizeof(int));
   return 0;
}
```

Passage d'un tableau de pointeurs en argument :

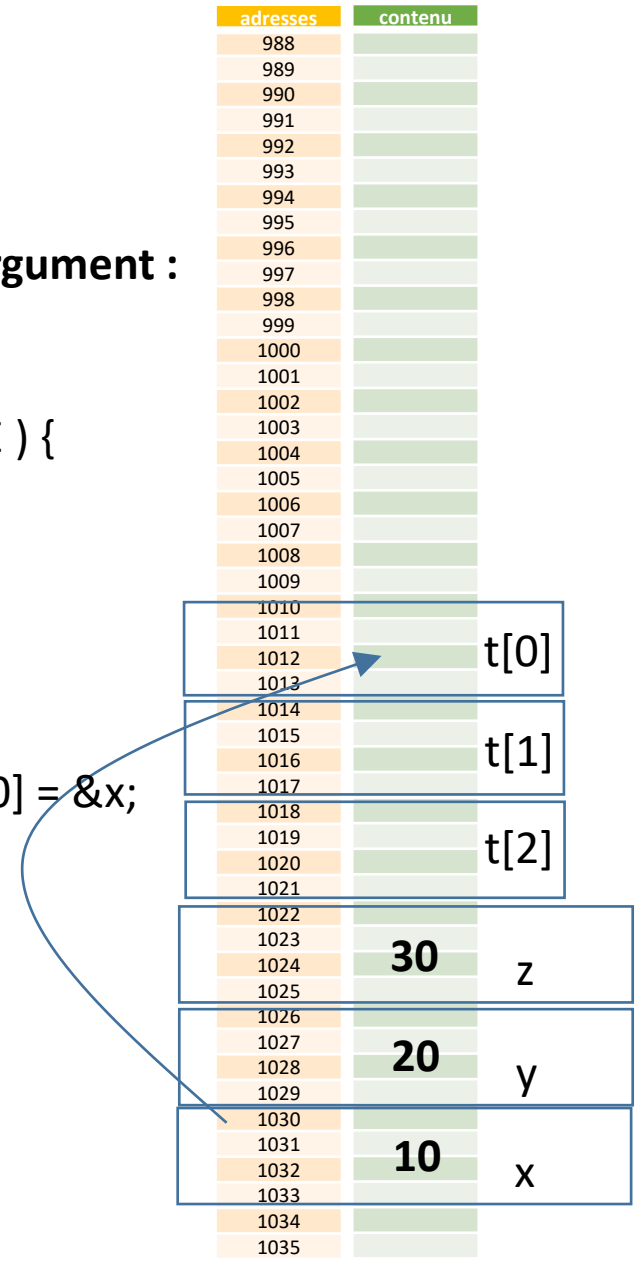
```
#include <stdio.h>
void AfficheTableau( int **tab, int TAILLE ) {
(⇔ void AfficheTableau( int *tab[], int TAILLE )
  int i;
  for (i=0; i<TAILLE; i++)
      printf("tab[%d]=%d \n", i, *tab[i]);
}

int main () {
   int *t[3];  /*déclaration d'un tab. de pointeurs */
   int x=10, y=20, z=30;
   t[0] = &x; t[1]=&y; t[2]=&z;
   AfficheTableau(t, sizeof(t)/sizeof(int));
   return 0;
```

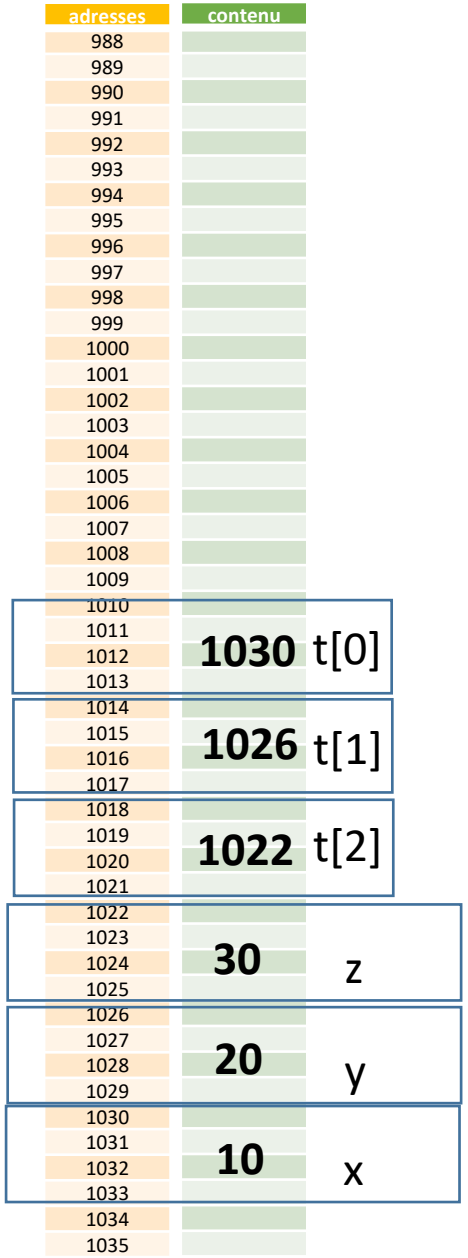
Tableau de pointeurs : exemple sur des int

Passage d'un tableau de pointeurs en argument :

```
#include <stdio.h>
void AfficheTableau( int **tab, int TAILLE ) {
  int i;
  for (i=0; i<TAILLE; i++)
      printf("tab[%d]=%d \n", i, *tab[i]);
}

int main () {
   int *t[3], x=10, y=20, z=30;

   t[0] = &x;
   t[1] = &y;
   t[2]=  &z;
   AfficheTableau(t, sizeof(t)/sizeof(int));
   return 0;
}
```

t[0] = &x;





Déclaration des chaines de caract : attention

```
#include <stdio.h>
#include <string.h>

int main () {
  char *c1="Bonjour1";
  char c2[27]="Bonjour2";
  int i=10;

  printf("c1=%s, adresse c1=%p \n", c1, c1);
  printf("c2=%s, adresse c2=%p \n", c2, c2);
  printf("i=%d, adresse i =%p \n", i, &i);

  strcpy(c1, "abcdefghijklmnopqrstuvwxyz");
  strcpy(c2, "abcdefghijklmnopqrstuvwxyz");
  return 0;
}
```

c1=Bonjour1,
adresse c1=0x80485a0
c2=Bonjour2,
adresse c2=0xbfebf2a5
i=10,
adresse i =0xbfebf2a0

non,
car c1 est alloué dans une zone mémoire non modifiable

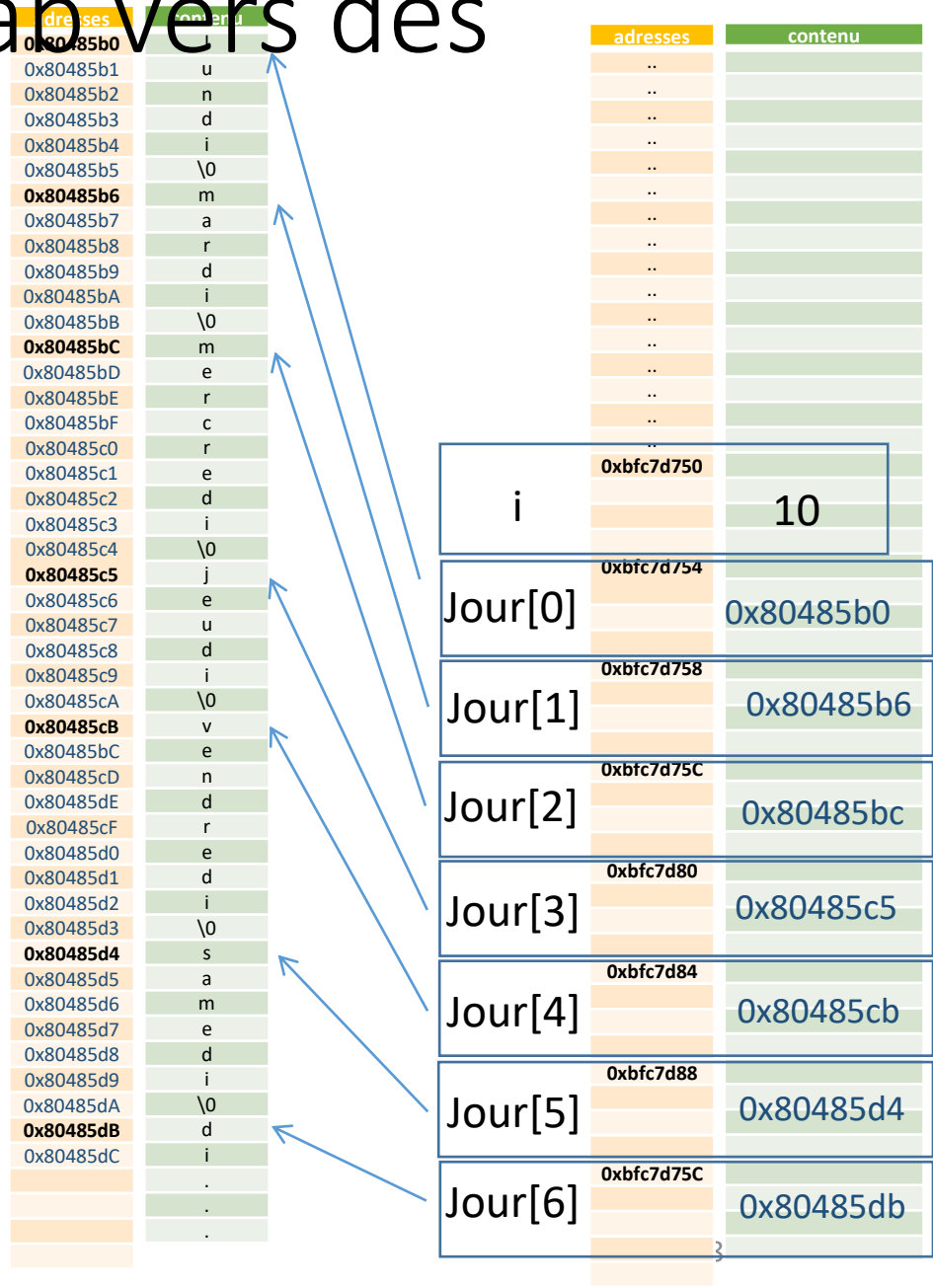
```
#include <stdio.h>
#include <string.h>

int main () {
  const char *c1="Bonjour1";
  char c2[]="Bonjour2";
  int i=10;

  printf("c1=%s, adresse c1=%p \n", c1, c1);
  printf("c2=%s, adresse c2=%p \n", c2, c2);
  printf("i=%d, adresse i =%p \n", i);

  strcpy(c1, "abcdefghijklmnopqrstuvwxyz"); /*Erreur compile grace à const */
  strcpy(c2, "abcdefghijklmnopqrstuvwxyz");
  return 0;
}
```

const indique au compilateur
qu'on ne doit pas essayer de modifier la variable

Tableau de pointeurs : exemple tab vers des chaines de caract.

```
#include <stdio.h>
int main () {
    int i=10;
    char *Jour[7]={"lundi","mardi",
    "mercredi","jeudi","vendredi","samedi","dimanche"};

    printf("&i=%p \n", &i);
    printf("Jour=%p \n", Jour);
    for (i=0;i<7;i++) {
     printf("Jour[%d]: %s ", i, Jour[i]);
     printf(", Jour[%d]: %p \n", i, Jour[i]);
    }
  return 0;
}
```

```
&i=    0xbfc7d750
Jour=0xbfc7d754
Jour[0]: lundi ,        Jour[0]: 0x80485b0
Jour[1]: mardi ,        Jour[1]: 0x80485b6
Jour[2]: mercredi ,  Jour[2]: 0x80485bc
Jour[3]: jeudi ,        Jour[3]: 0x80485c5
Jour[4]: vendredi ,  Jour[4]: 0x80485cb
Jour[5]: samedi ,    Jour[5]: 0x80485d4
Jour[6]: dimanche , Jour[6]: 0x80485db
```



tab. pointeurs vs tab char 2D

```
#include <stdio.h>
int main () {
   int i;
   char * Noms[3]={"Paul","Eric","Martin"};
   /* (17 octets utilisés, non modifiables) */
   printf("Noms=%p \n", Noms);
   for (i=0;i<3;i++) {
     printf("Noms[%d]: %s %p ", i, Noms[i], Noms[i]);
   }
  return 0;
}
```

```
Noms=0xbfc89df4
Noms[0]: Paul    0x8048580
Noms[1]: Eric    0x8048585
Noms[2]: Martin 0x804858a
```

```
#include <stdio.h>
int main () {
   int i;
   char Noms[3][10]={"Paul","Eric","Martin"};
   /* (30 octets utilisés, mais modifiable) */
   printf("Noms=%p \n", Noms);
   for (i=0;i<3;i++) {
     printf("Noms[%d]: %s %p ", i, Noms[i],
   Noms[i]);
   }
  return 0;
}
```

```
Noms=0xbf9af442
Noms[0]: Paul    0xbf9af442
Noms[1]: Eric    0xbf9af44c
Noms[2]: Martin 0xbf9af456
```

Application : arguments du main

Il est possible de passer des arguments sur la ligne de commande au moment du lancement du programme dans le terminal :

ex: ./programme toto 10

- Le nombre d'arguments est stocké dans une variable "int argc" *(ici vaut 3 car le nom du pgm compte comme un argument)*
- Ces arguments sont stockés sous forme de chaine de caractères dans un tableau de pointeurs : char *argv[ ]

argv[0] : pointe vers le nom du programme (ici "programme")

argv[1] : pointe vers le 1er argument (ici "toto")

argv[2] : pointe vers le 2nd argument (ici "10", attention c'est une chaine de caractères, cad '1' et '0' et '\0'

Arguments du main (ligne de commande)

```
#include<stdio.h>

int main( int argc, char *argv[]) {
    printf("Le nombre d'arguments est %d \n", argc);

     printf("L'argument 0 est %s \n", argv[0]);

     if (argc>=2)
       printf("L'argument 1 est %s \n", argv[1]);

     if (argc>=3)
       printf("L'argument 2 est %s \n", argv[2]);
  return 0;
}
```

```
esiee@debian:~/testc$ gcc -Wall test1.c -o test

esiee@debian:~/testc$ ./test
Le nombre d'aguments est 1
L'argument 0 est test

esiee@debian:~/testc$ ./test toto
Le nombre d'aguments est 2
L'argument 0 est test
L'argument 1 est toto

esiee@debian:~/testc$ ./test toto 10
Le nombre d'arguments est 3
L'argument 0 est test
L'argument 1 est toto
L'argument 2 est 10
```

Arguments du main (ligne de commande)

Attention tous les arguments sont des chaînes de caractères, on ne peut donc pas écrire :

```
int main(int argc, char *argv[]){
    int var;
    var = argv[1];   /* car argv[] pointe vers une chaine de caract */
```

Il faut convertir la chaine de caractère en un entier : fonction atoi

```
    var = atoi(argv[1]);  /*ne pas oublier le bon #include...*/
```

ATOI(3) Linux Programmer's Manual ATOI(3)

NAME
atoi, atol, atoll, atoq - convert a string to an integer

SYNOPSIS
#include <stdlib.h>

int atoi(const char *nptr);

Pointeur vers fonctions

Le type de pointeur void * : il peut pointer vers n'importe quel type

Exemple

```
int a;
char b;
void *p;
p=&a;  //ne génère aucun warning
p=&b; // bien que les variables pointées soient de type différents.
```

Appel de fonction : rappel assembleur

```
void Init_tab (int t[]){
     int i;
          for (i=0;i<100;i++) {
          t[i]=i;
    }
}
```

```
__semihosting_library_function:
0x00000206   MOV    r1,r0
0x00000208   MOVS   r0,#0x00
0x0000020A   B      0x00000212
0x0000020C   LSLS   r2,r0,#2
0x0000020E   STR    r0,[r1,r2]
0x00000210   ADDS   r0,r0,#1
0x00000212   CMP    r0,#0x64
0x00000214   BLT    0x0000020C
0x00000216   BX     lr
```

adresse de la fonction

```
int main() {
  int a=10,b=0;
    int tab[100];
    Init_tab (tab);
    b=a*tab[1];
    return 0;
}
```

```
main:
0x00000228   PUSH   {r4,lr}
0x0000022A   SUB    sp,sp,#0x190
0x0000022C   MOVS   r3,#0x0A
0x0000022E   MOVS   r4,#0x00
0x00000230   MOV    r0,sp
0x00000232   BL.W   __semihosting_library_function (0x00000206)
0x00000236   LDR    r0,[sp,#0x04]
0x00000238   MULS   r0,r3,r0
0x0000023A   MOV    r4,r0
0x0000023C   MOVS   r0,#0x00
0x0000023E   ADD    sp,sp,#0x190
0x00000240   POP    {r4,pc}
0x00000242   MOVS   r0,r0
```

Pointeur vers fonctions : affectation

Chaque fonction a donc une adresse :

- on peut obtenir l'adresse de cette fonction avec & devant le nom d'une fonction
- on peut stocker l'adresse d'une fonction dans un pointeur de type void*

```
#include<stdio.h>
void Affiche () {  printf("Hello");}

int main() {
   void *pointeur1;
   Affiche();
  pointeur1 = &Affiche;                    /*⇔ pointeur1 = Affiche; */
   return 0;
}
```

Nom de la fonction ⇔ son adresse (le & n'est pas nécessaire)

pour pouvoir appeler la fonction depuis son pointeur il faut donner des info. supplémentaires...

Pointeur vers fonctions : affectation

Problème : les arguments de la fonction et le type retourné

⇔comment le compilateur peut-il vérifier...?

➔la déclaration du pointeur dépend des arg. et du type retourné

type retourné	Sans argument	Avec arguments
aucun	void (*pointeur_fonc)(void);	void (*pointeur_fonc)(int);
int	int (*pointeur_fonc)(void);	int (*pointeur_fonc)(int, char);
char	char (*pointeur_fonc) (void);	char (*pointeur_fonc)(int *, char, short);
char *	char * (*pointeur_fonc)(void);	
etc	etc	etc

Pointeur vers fonctions : affectation

```
#include<stdio.h>
void Affiche () {  printf("Hello");}
int add(int a,int b) { return a+b;}

int main() {
   void (*pointeur1)(void);
   int (*pointeur2)(int, int);

   pointeur1 = Affiche;
   pointeur2 = add;
   return 0;
}
```

Pointeur vers fonctions : utilisation

Pour appeler une fonction à partir de son pointeur : il suffit d'écrire le nom du pointeur précédé de * et suivi des arguments :

```
/* Sans arguments */
#include<stdio.h>
void Affiche () {  printf("Hello");}
int main() {
   void (*pointeur1)(void);   /* decl. */
   pointeur1 = Affiche;        /* affec. */
   (*pointeur1) ();            /* utilisa. */
   return 0;
}
```

```
/*Avec arguments */
#include<stdio.h>
int add(int a,int b) { return a+b;}
int main() {
    int x=10,y=20, z;
    int (*pointeur2)(int, int);     /*déclaration*/
    pointeur2 = add;                /* affectation */
   z= (*pointeur2) (x,y);       /* utilisation */
    printf("z=%d", z);
   return 0;
}
```

Tableau de pointeurs vers fonctions

```
#include<stdio.h>
void Affiche1 () { printf("Hello \n");}
void Affiche2 () { printf("ESIEE \n");}

int main() {
  void (*pointeurs[2])(void);

  pointeurs[0] = Affiche1;
  pointeurs[1] = Affiche2;

  (*pointeurs[0]) ();
  (*pointeurs[1]) ();
  return 0;
}
```

Permet par exemple d'éviter de nombreux switch/case :

```
switch (i) {
  case 0 : Affiche1(); break;
  case 1 : Affiche2(); break;
  etc...
}
```

⇔ (*pointeurs[i]) ();

Entrées-Sorties Fichier

Chapitre VI

Gestion de fichier : principes

Objectif : rendre vos programmes capables de lire/écrire dans des fichiers (texte, images, sons, valeurs de mesures etc)

3 étape ⇔ 3 fonctions C

1. ouvrir le fichier : **fopen**
2. accéder au contenu du fichier (lire et/ou écrire dedans) : **fscanf** / **fprintf**
3. fermer le fichier : **fclose**

Au moment de l'ouverture avec fopen on indique :

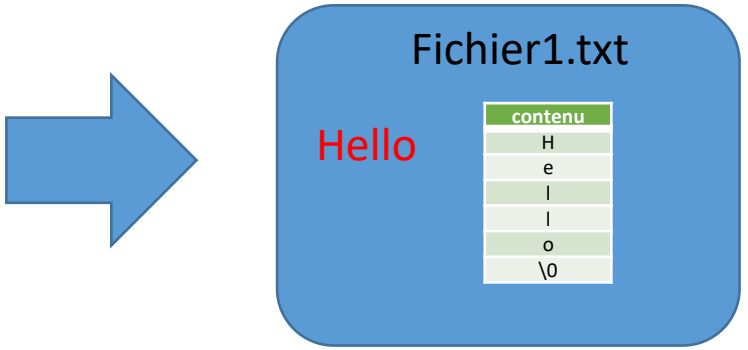
- le nom du fichier à ouvrir : une chaine de caractère
- ET le type d'accès au moyen d'une chaine de caractères :
  - "r" pour lecture uniquement (read),
  - "w" pour écriture (et écrasement si existe déjà),
  - "a" (pour append) n'écrase pas si il existe mais ajoute à la fin du fichier

Gestion de fichier : écriture

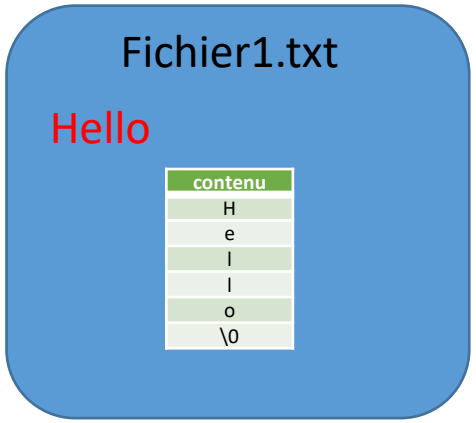
Il est possible d'accéder simultanément à plusieurs fichiers : donc chaque fonction à besoin de savoir sur quel fichier elle agit ⇔ donc fopen retourne un pointeur (de type FILE) vers le fichier qui est ouvert, il faut ensuite passer ce pointeur aux fonctions fprintf, fscanf et close

Attention : si fopen retourne le pointeur NULL ⇔ le fichier n'a pas été ouvert (il n'existe pas, non incorrect, problème de droits d'accès dans le répertoire...) DONC TOUJOURS TESTER LA VALEUR RETOURNEE

```
#include <stdio.h>
int main(){
  char mot[30]="Hello";              /*pour stocker le mot que l'on va lire */
  FILE *f;                           /*pointeur vers le fichier ouvert */
  f=fopen("Fichier1.txt","w");       /* ouverture du fichier en mode ecriture (w) */
  if (f==NULL) {
     printf("Le fichier n'a pas pu être créé. Fin du programme");
     return 1;
  }
  fprintf(f, "%s", mot);             /*écriture d'1 mot dans le fichier  ouvert*/
  fclose(f);                         /* fermeture du fichier */
  return 0;
}
```



Gestion de fichier : lecture




```
int main(){
 char mot[30];                          /*pour stocker le mot que l'on va lire */
 FILE *f;                               /*pointeur vers le fichier ouvert */
 f=fopen("Fichier1.txt","r");           /* ouverture du fichier en mode lecture (r) */
 if (f==NULL) {
    printf("Le fichier n'a pas pu être ouvert. Fin du programme");
    return 1;
  }
 fscanf(f, "%s", mot);                  /*lecture d'1 mot du fichier  ouvert*/
 printf("Le premier mot lu dans le fichier est %s",mot);
 fclose(f);                             /* fermeture du fichier */
 return 0;
}
```

Gestion de fichier : lecture

- fscanf considère comme des mots chaque séquence de code ASCII qui ne comprend ni le caractère espace, ni un retour chariot (\n), ni une tabulation (\t)

Exemple : fichier.txt

Ceci est un texte dont chaque mot est séparé par un espace, une tabulation ou un retour à la ligne

C	67
e	101
c	99
i	105
	32
e	101
s	115
t	116
	32
u	117
n	110
	32
t	116
e	101
x	120
t	116
	32
d	100
o	111
n	110
t	116
	32
c	99
h	105

a	97
q	113
u	117
e	101
\n	13
m	109
o	111
t	116
	32
e	101
s	115
t	116
	32
s	115
é	136
p	112
a	97
r	114
é	136
	32
p	112
a	97
r	114

Gestion de fichier : lecture

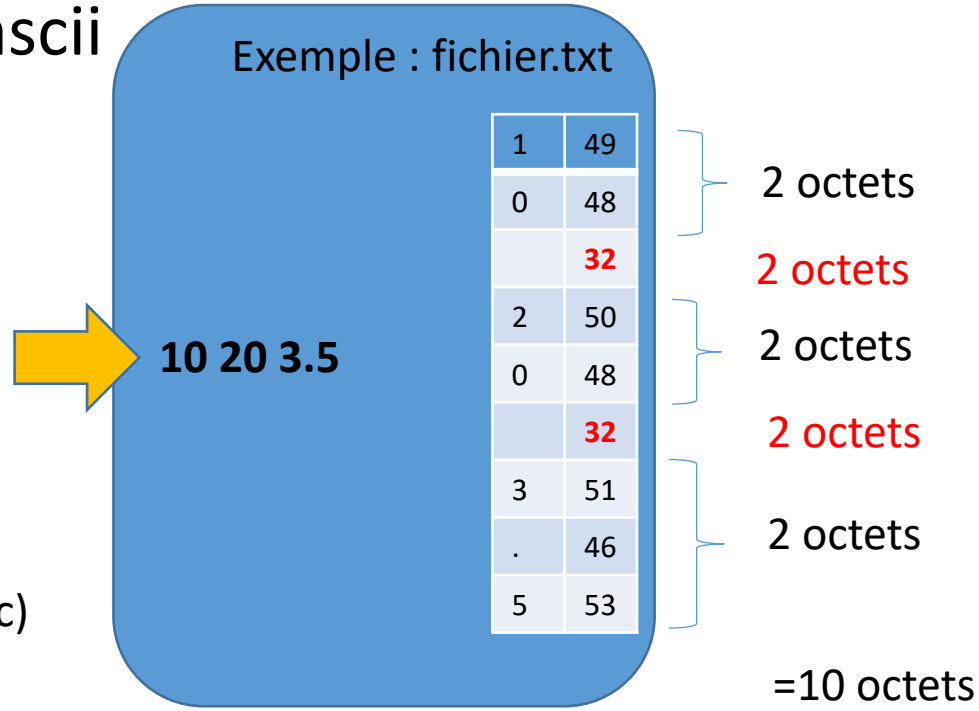
- On peut aussi lire/écrire
  - des entiers (avec %d)
  - flottants (avec %f)

⇔ ils sont stockées sous forme de caractères ascii

- Inconvénients : ils occupent alors bcp de place

➔passer en mode binaire

```
char a=10, b=20;
float c=3.5;
...
fprintf( f, "%d", a);
fprintf( f, "%d", b);
fprintf( f, "%f", c);

2*sizeof(a)+ sizeof(c)
=2+4 = 6 octets
```



Localisation du fichier : chemin d'accès

```
#include<stdio.h>
int main(){
 char mot[30];                          /*pour stocker le mot que l'on va lire */
 FILE *f;                               /*pointeur vers le fichier ouvert */
 f=fopen("../../NomFichier","r");       /* ouverture du fichier en mode lecture (r) */
 if (f==NULL) {
    printf("Le fichier n'a pas pu être ouvert. Fin du programme");
    return 1;
  }
 fscanf(f, "%s", mot);                  /*lecture d'1 mot du fichier  ouvert*/
 printf("Le premier mot lu dans le fichier est %s",mot);
 fclose(f);                             /* fermeture du fichier */
 return 0;
}
```

ou encore

```
f=fopen("c:/programmesC/NomFichier","r")
```

Gestion de fichier : position dans le fichier

Question : comment lire le second mot du fichier ?

La structure FILE maintient la position courante dans le fichier

A chaque lecture/écriture, la position courante dans le fichier est mise à jour de façon à ce que le prochain accès permette de lire l'élément suivant.

Détection de fin de fichier : feof DANGER

Comment savoir si on a atteint la fin de fichier ?

La fonction feof (Fin End Of File) retourne 1 si la fin de fichier est atteinte

Prototype : **int feof (FILE *stream);**

NE PAS UTILISER

```
#include <stdio.h>
int main(){
 char mot[30];                    /*pour stocker le mot que l'on va lire */
 FILE *f;                         /*pointeur vers le fichier ouvert */
 f=fopen("Fichier1.txt","r");     /* ouverture du fichier en mode lecture (r) */
 if (f==NULL) {printf("Le fichier n'a pas pu être ouvert. Fin du programme\n"); return 1;}
while ( feof(f)==0) {             /* Tant qu'on a pas atteint la fin du fichier */
   fscanf(f, "%s", mot);          /*lecture d'1 mot du fichier  ouvert*/
   printf("mot lu=%s \n",mot);
}
 fclose(f);  return 0;
}
```

ATTENTION DANGER : feof utilise le résultat du dernier accès

Détection de fin de fichier : fscanf

- La valeur retournée par la fonction fscanf permet aussi de savoir si la fin de fichier est atteinte : vaut EOF si fin de fichier atteinte

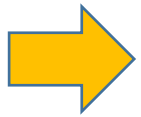
```
#include <stdio.h>
int main(){
  char mot[30];                    /*pour stocker le mot que l'on va lire */
  FILE *f;                         /*pointeur vers le fichier ouvert */
  f=fopen("Fichier1.txt","r");     /* ouverture du fichier en mode lecture (r) */
  if (f==NULL) {
    printf("Le fichier n'a pas pu être ouvert. Fin du programme\n");
    return 1;
  }
  while (fscanf(f, "%s", mot) != EOF) {   /* Tant que la lecture n'atteint pas la fin du fichier */
    printf("mot lu=%s \n",mot);
  }
  fclose(f);                       /* fermeture du fichier */
  return 0;
}
```

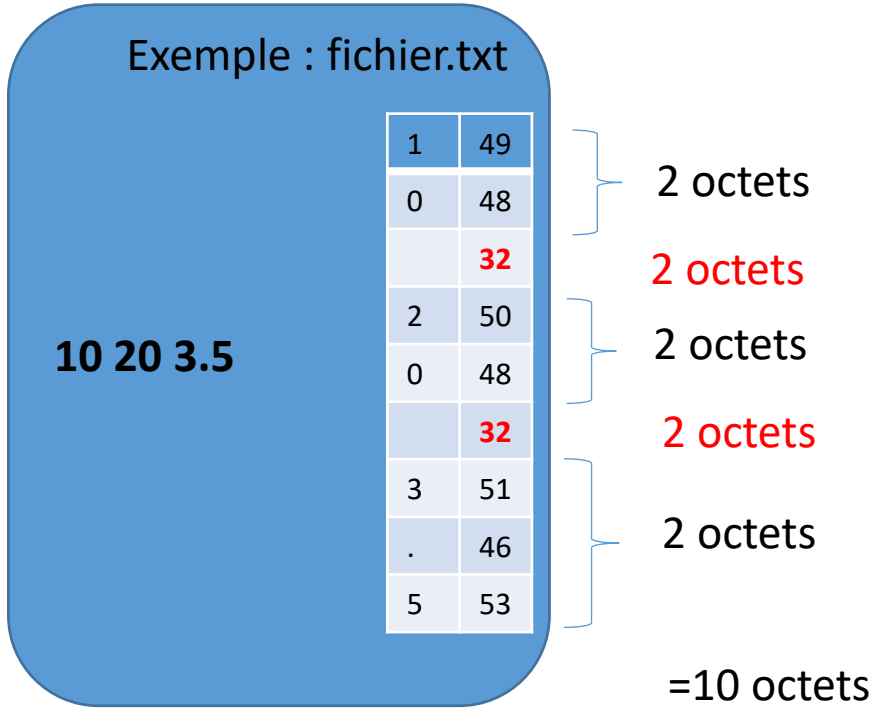
Fichier binaires

- Par défaut les fichiers sont ouvert en mode "ASCII" (dit aussi mode "Texte") :

Mode texte

```
#include <stdio.h>
int main(){
  char a=10, b=20;
  float c=3.5;
  FILE *f;
  f=fopen("Fichier1.txt","w");
  if (f==NULL) {
     printf("Le fichier n'a pas pu être ouvert.");
     return 1;
  }
  fprintf(f, "%d",a);
  fprintf(f, "%d",b);
  fprintf(f, "%f",c);
 fclose(f);
  return 0;
}
```





209

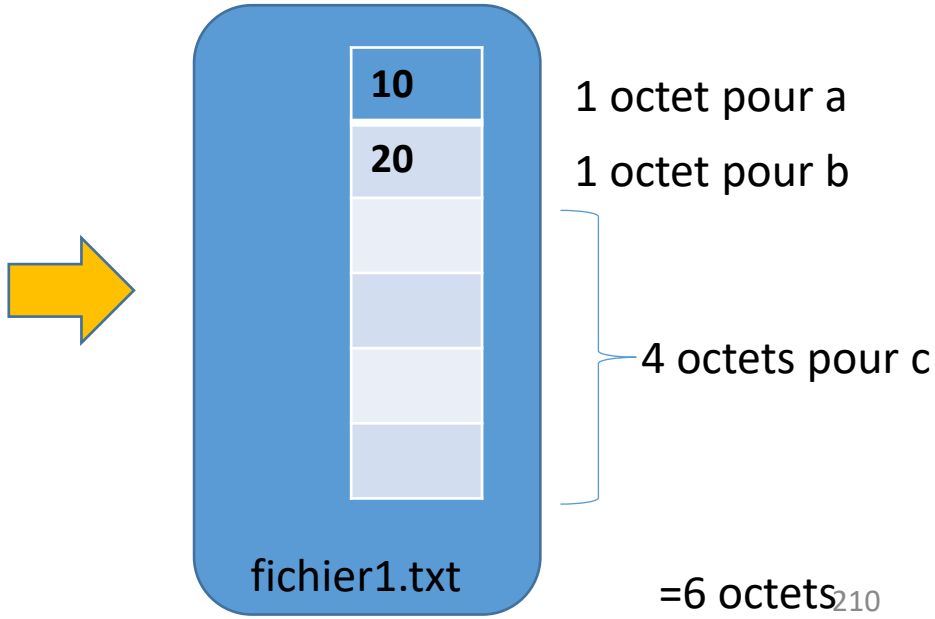
Fichiers binaires

- on peut aussi ouvrir le fichier en mode "binaire" pour gagner plus de place : utilisation de fwrite et fread

#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite (const void *ptr, size_t size, size_t nmemb, FILE *stream);

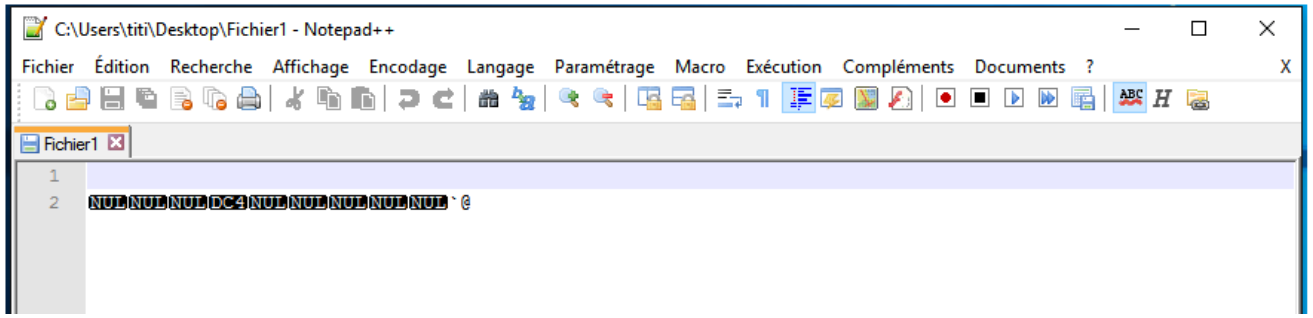
Mode binaire

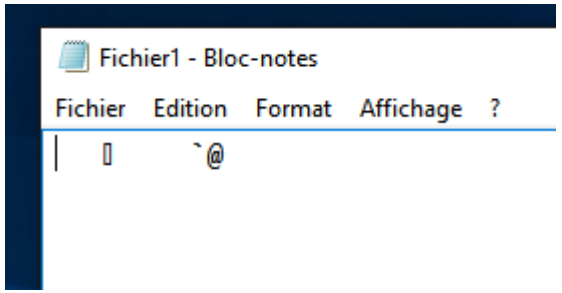
```
#include <stdio.h>
int main(){
  int a=10, b=20; float c=3.5;
  FILE *f;
  f=fopen("Fichier1.txt","wb");
  if (f==NULL) {
     printf("Le fichier n'a pas pu être ouvert. Fin du programme");
     return 1;
  }
  fwrite(&a, sizeof(a),1,f);
  fwrite(&b, sizeof(b),1,f);
  fwrite(&c, sizeof(b),1,f);
  fclose(f);
  return 0;
}
```



Fichiers binaires vs fichiers textes

- Les fichiers textes peuvent être ouvert avec n'importe quel éditeur
- Ouvrir un fichier texte avec un éditeur de texte standard risquera d'afficher n'importe quoi puisque l'éditeur affichera le code ASCII de chaque octet trouvé dans le fichier...





➔ il faut utiliser un éditeur spécifique pour le binaire qui en général affiche le contenu du fichier sous forme de nombres en hexadécimal

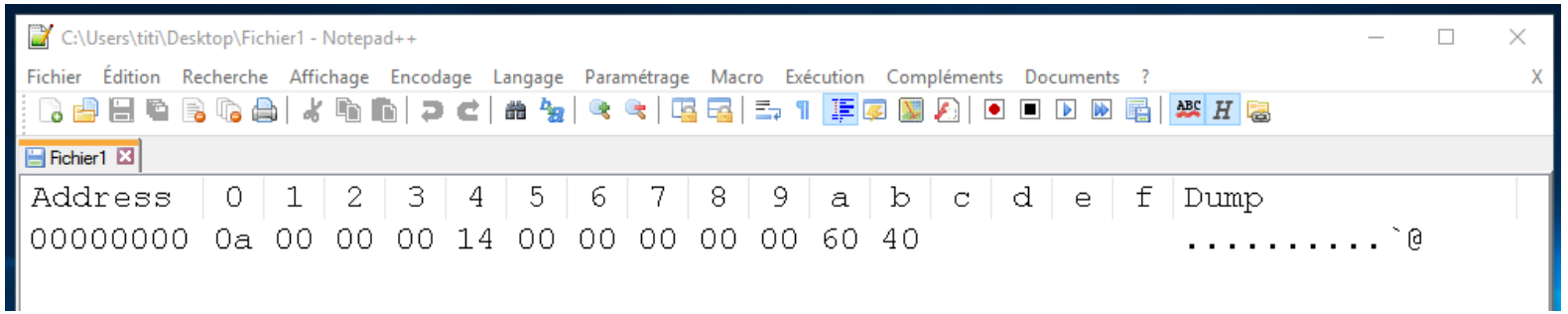
exemple hexdump sous linux, ou notepad++ avec l'extension hex-editor

Fichiers binaires : ouverture avec éditeur hexa

```
#include <stdio.h>
int main(){
  int a=10, b=20; float c=3.5;
  FILE *f;
  f=fopen("Fichier1","wb");
  if (f==NULL) {
     printf("Le fichier n'a pas pu être ouvert");
     return 1;
  }
  fwrite(&a, sizeof(a),1,f);
  fwrite(&b, sizeof(b),1,f);
  fwrite(&c, sizeof(b),1,f);
  fclose(f);
  return 0;
```

```
esiee@debian:~/testc$ hexdump Fichier1
0000000 000a 0000 0014 0000 0000 4060
000000c
```

octets: n° 0 1 2 3 4 5 6 7 8 9 a b



VII-Organisation de la mémoire

Organisation de la mémoire

- Zone mémoire dédiée au programme
- Zone mémoire dédiée au constante en lecture uniquement
- Zone mémoire dédiée au variables globales

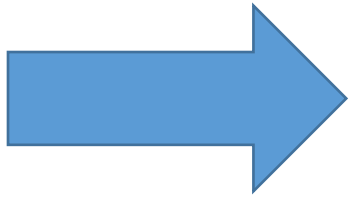
- Zone mémoire dédiée au variables locales de chaque fonction : la pile

- Nouvelle zoné mémoire : le tas

Tas et Allocation dynamique

Nous avons vu qu'une fonction ne pouvait créer un tableau et le retourné à l'appelant puisqu'il était alloué sur la pile :

```
#include <stdio.h>
#define TAILLE 7

int* CreerTab() {
      int t[TAILLE]={11,22,33,44,55,66,77};
      return t; //⇔ return &t[0];
}

int main () {
              int *tab, i;
    tab=CreerTab();
              for (i=0; i<TAILLE; i++)
       printf("tab[%d]=%d \n", i, tab[i]);
   return 0;
}
```



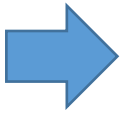
il existe une solution à ce problème :
allouer le tableau ailleurs que sur la pile
cette zone s'appelle le TAS (heap en anglais)

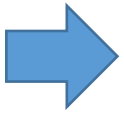
Tas et Allocation dynamique

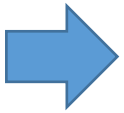
- Principe :
  - déclarer un pointeur local
  - appeler la fonction d'allocation dynamique malloc qui va allouer un espace mémoire dans le tas :
    - il faut lui donner en argument : le nombre d'octets à réserver
    - elle retourne un pointeur vers le début de laa zone réservée
  - on stocke la valeur retournée par malloc dans le pointeur local
  - on utilise ce pointeur local pour lire/écrire dans la zone du tas

  - quand on a plus besoin de cette zone allouée on libère explicitement en utilisant la fonction free

Tas et Allocation dynamique : malloc / free



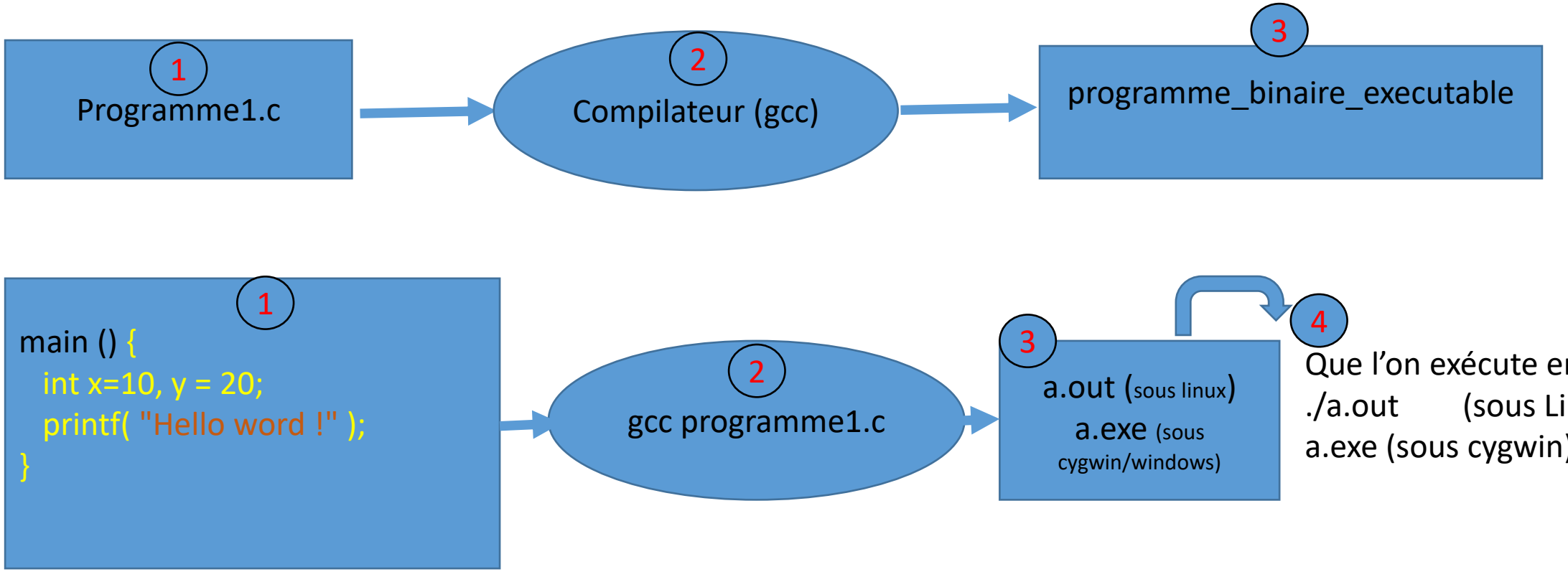




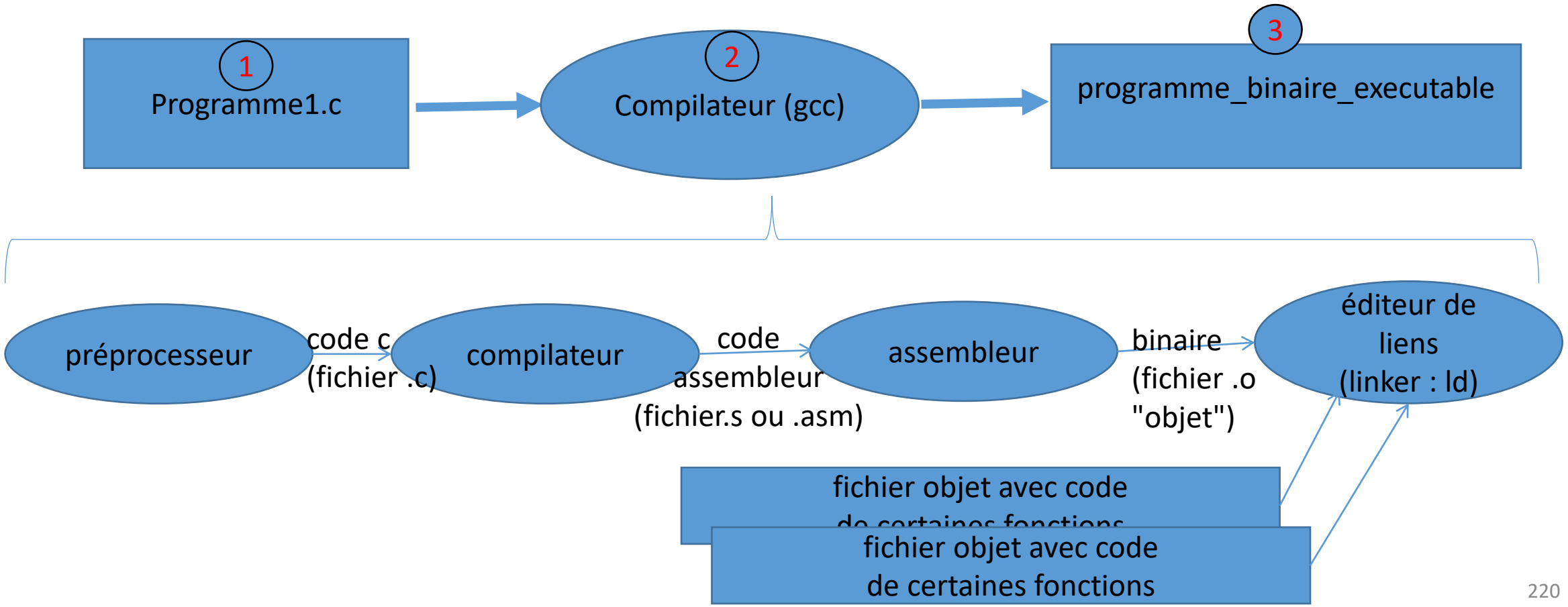
```
#include <stdio.h>
#include <stdlib.h>
int* CreerTab(int taille) {
      int *t;
      t=malloc(taille*sizeof(int) );
      for (i=0;i<taille;i++)
         t[i]=i;
      return t;
}
int main () {
    int *tab, i;
    const int max=5;
    tab=CreerTab(max);
    for (i=0; i<max; i++)
        printf("tab[%d]=%d \n", i, tab[i]);
    free(tab); // pour liberer l'espace mémoire réservé dans le tas
    return 0;}
```

Chaîne de compilation

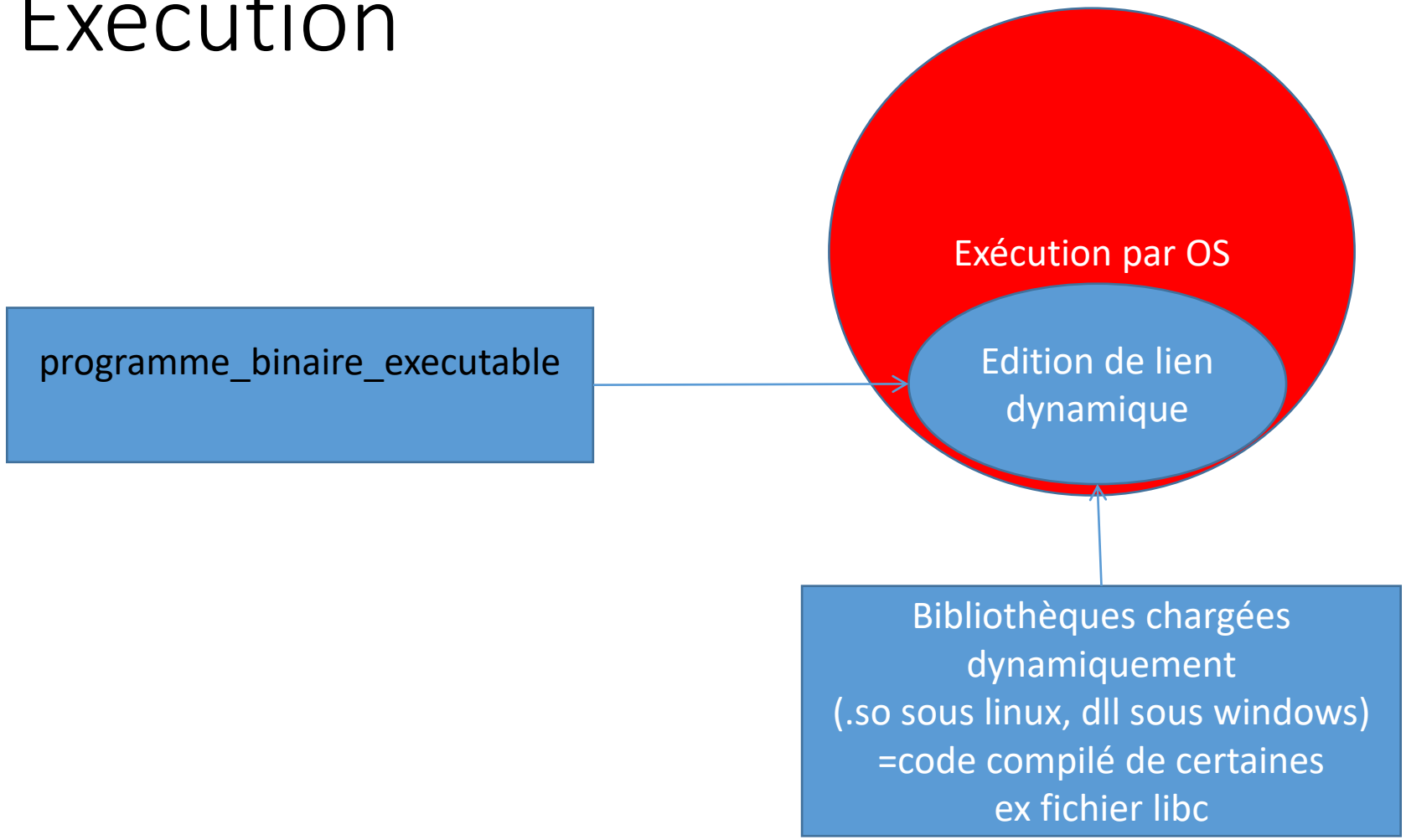
Chaine de compilation



Chaine de compilation



Exécution


programme_binaire_executable
Exécution par OS
Edition de lien dynamique
Bibliothèques chargées dynamiquement
(.so sous linux, dll sous windows)
=code compilé de certaines
ex fichier libc

Préprocesseur : symboles, macros

- Entrée : fichier C
- Sortie : fichier C dans lequel toutes les directives ont été traitées
  - Inclusion de fichiers #include <NomFichier> et #include "NomFichier"
  - Symboles : #define MAX 10 ➔ remplace MAX par 10 dans tous le fichier
  - Macros : #define carre(a) a*a
  - Symbole prédéfinis : __LINE__ __DATE__ __TIME__

- Il est possible de voir le fichier obtenu après passage dans le préprocesseur avec l'option –E de gcc

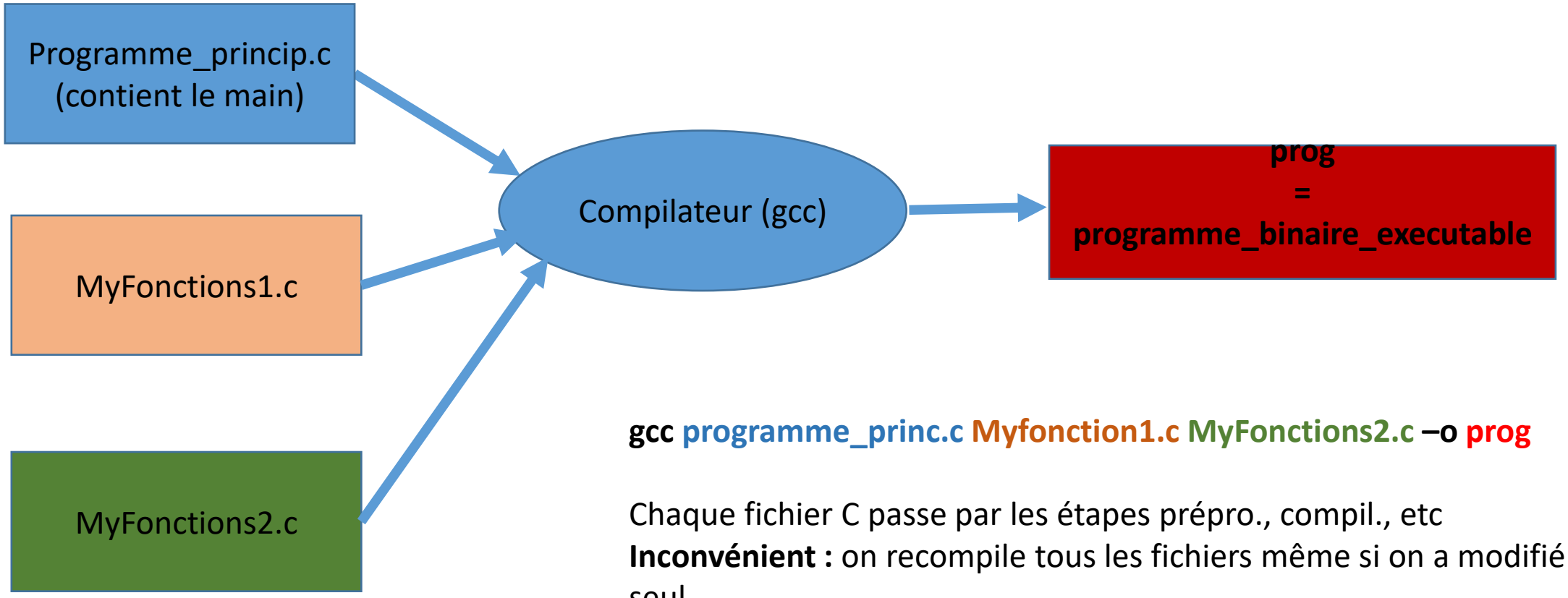
Compilateur

- Entrée : fichier c
- Sortie : fichier en assembleur
- Il est possible d'afficher le fichier assembleur produit avec l'option –S de gcc

Assemblage

- Entrée : fichier assembleur
- Sortie : fichier objet
- Il est possible d'arreter la chaîne de compilation après la fabrication du fichier objet avec l'option –c de gcc

Compilation multifichiers

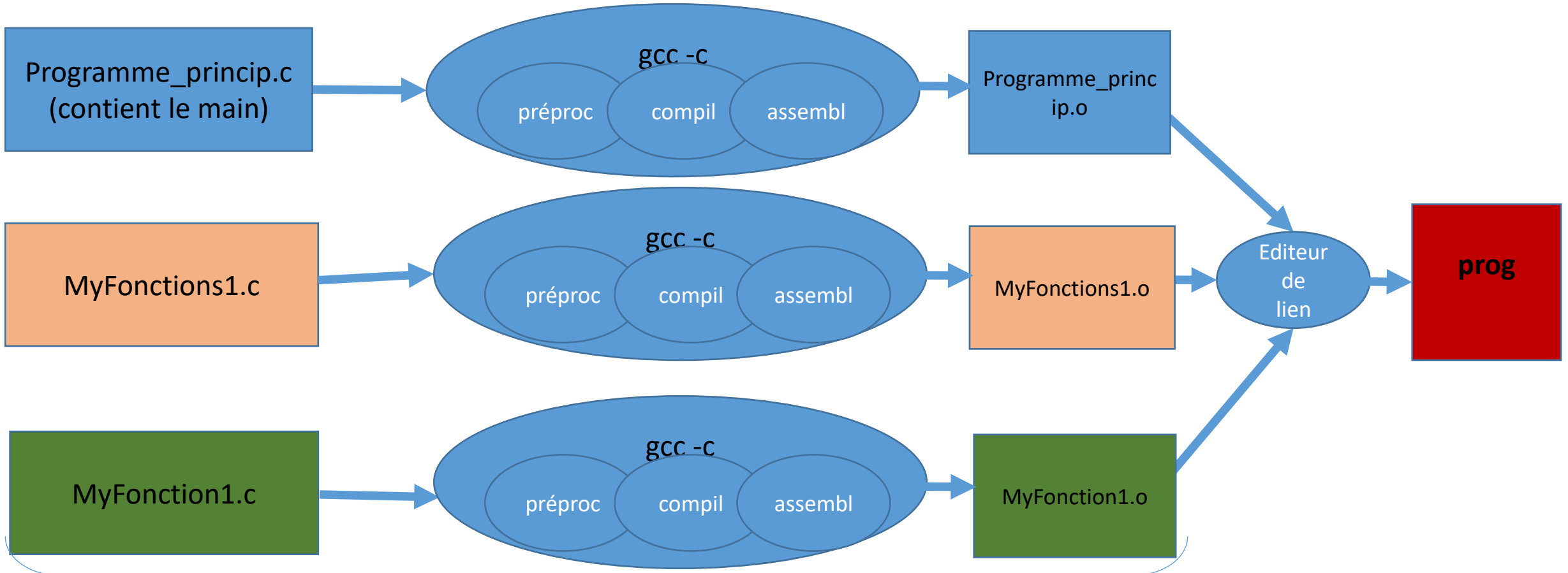


gcc programme_princ.c Myfonction1.c MyFonctions2.c –o prog

Chaque fichier C passe par les étapes prépro., compil., etc
Inconvénient : on recompile tous les fichiers même si on a modifié qu'un seul

(la durée de compilation peut atteindre plusieurs heure sur de gros projets!)

Compilation multifichiers : compil. séparée



1 gcc programme_princ.c -c

1 gcc Myfonction1.c -c

1 gcc MyFonctions2.c –c

2 gcc programme_princ.o Myfonction1.o MyFonctions2.o –o prog

Compilation multifichiers : compil. séparée

- La durée de compilation de gros projets peut atteindre plusieurs heures (par exemple noyau linux...)
- Avantage : on ne recompile que le fichier qui a été modifié, seule l'étape d'edition de lien est ensuite nécessaire (puisque les autres fichiers objets .o n'ont pas été modifiés)

➔ gain de temps important pour les gros projets !

Edition de liens dynamique

- Dans l'exemple précédent de compilation séparée l'exécutable obtenu contient le code de toutes les fonctions
- Avantage : tout est dans un fichier
- Inconvénient : si on utilise les fonctions dans plusieurs programmes différents ils seront inclus dans tous ces fichiers (occupation d'espace disques), ils seront chargés plusieurs fois en mémoire (occu. espace mémoire RAM)
- Solution : bibliothèque chargée dynamiquement et partagée
  - permet en plus de faire évoluer les fonctions sans recompiler le prog. princip.
  - Inconvénient : il faut distribuer ces biblio. avec l'executable (....dll windows)..

Pour aller plus loin

Champs de bits

Format particuliers de printf

%5.3

Mots clef : "volatile", "register"

Fonctions à nombre d'arguments variable

- Exemple la fonction printf

Structures de données : liste chaînées

- Utilisation de pointeurs + structures + allocation dynamique

Bibliographie