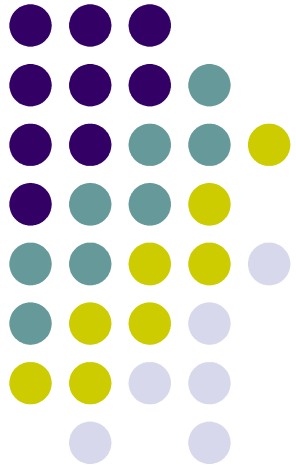
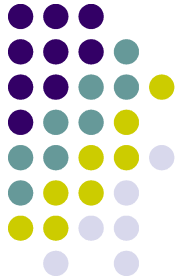
Pointers in the pocket

Introduction en douceur des concepts mystiques du fonctionnement des pointeurs en C

Cahier de coloriage crée par Vlad TRIFA
Octobre 2004, EPFL





DISCLAIMER

Ce texte est destiné à des enfants en bas âge afin de leur apprendre la programmation par pointeurs.

Ne vous étonnez donc pas, de la simplicité et de la clarté des explications, vous ne trouverez rien de plus simple sur ce sujet ailleurs sur Internet.

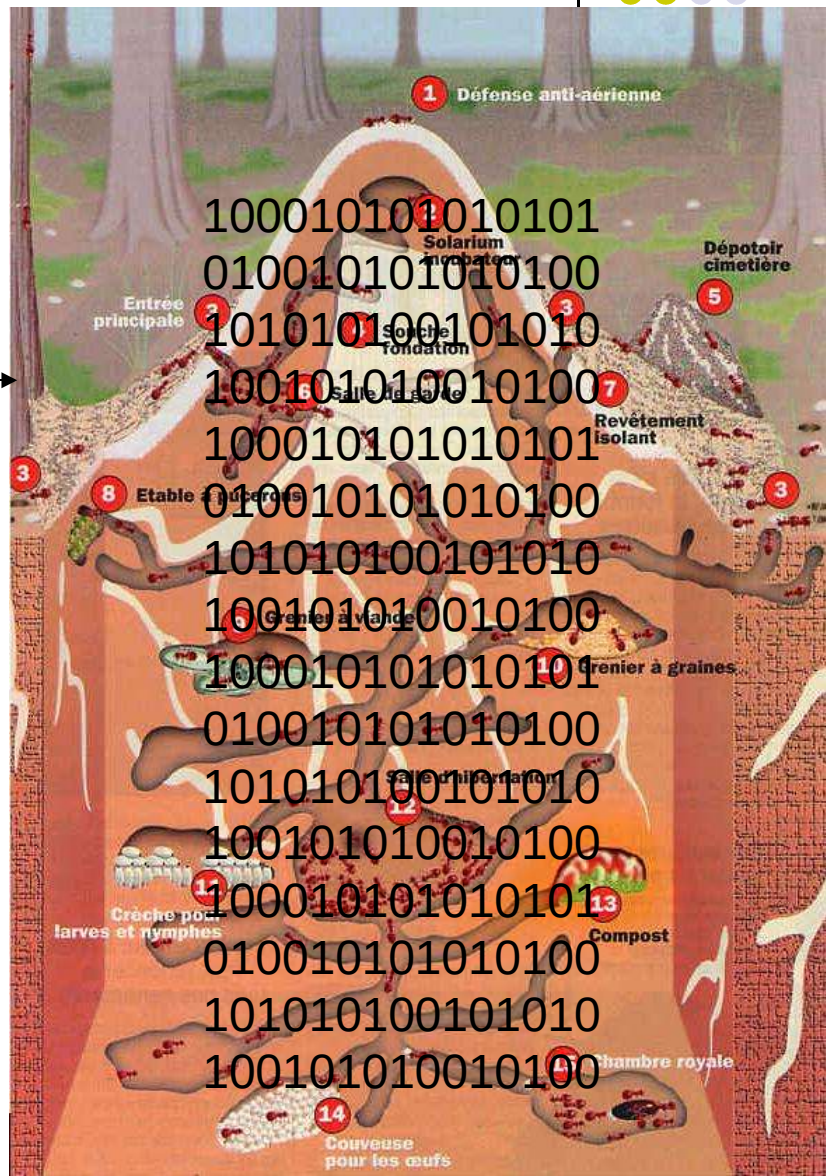
Cependant si vous êtes homme d'affaire, ou que sais-je d'autre et que vous devez maîtriser les pointeurs en 1 heure chrono, alors ce texte est fait pour vous aussi.

Ayez bien du plaisir avec ce document.

Données?

TOUTES les données sans exception sont stockées dans une gigantesque fourmilière de 1 et de 0, symboles qu'on appellera communément **bit**.

BIT = Binary Information Unit

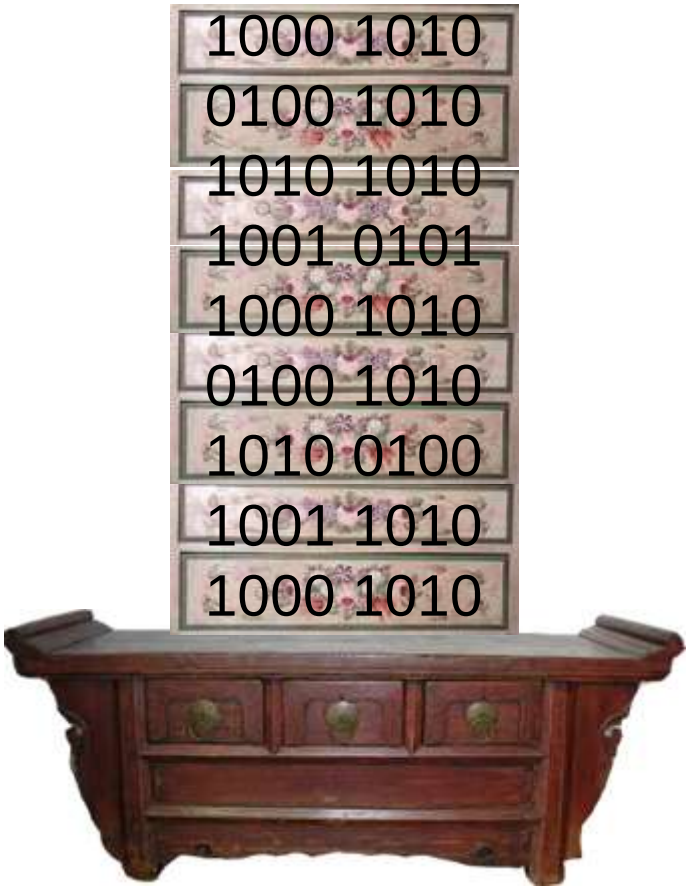


Organisation de la mémoire

Un jour, on s'est dit qu'il va falloir classer tout ça.

Pour cela, on a décider de ranger tous ces 0 et 1 dans une très grande armoire composée de millions de tiroirs. Pour des raisons d'efficacité, on utilise uniquement des tiroirs qui ne peuvent contenir que 8 bits chacun.

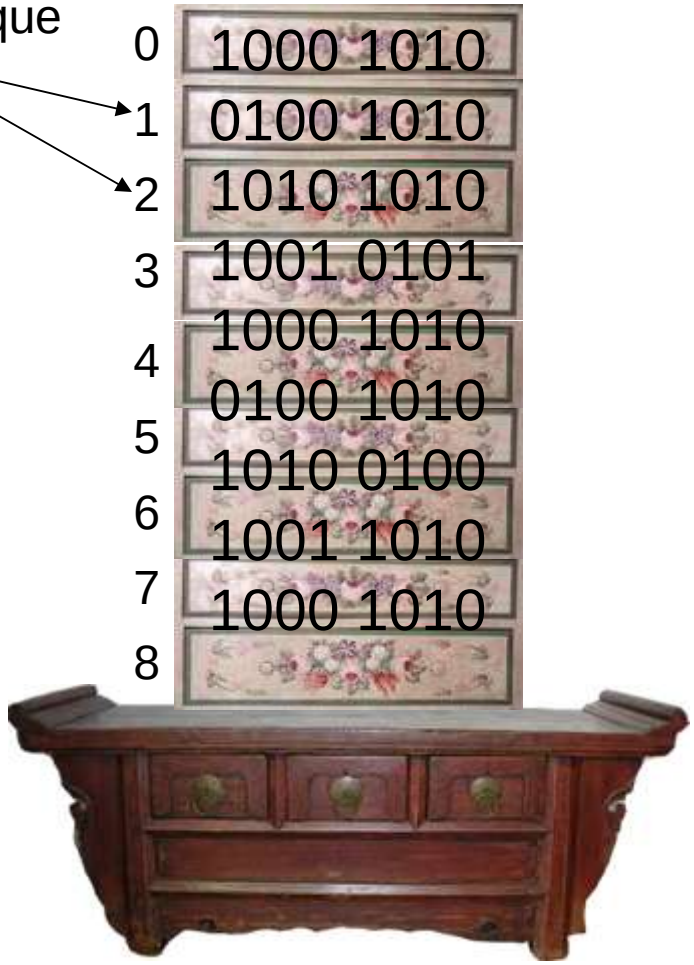
L'avantage est qu'on peut utiliser plusieurs tiroirs pour stocker une seule et même information.



Organisation de la mémoire

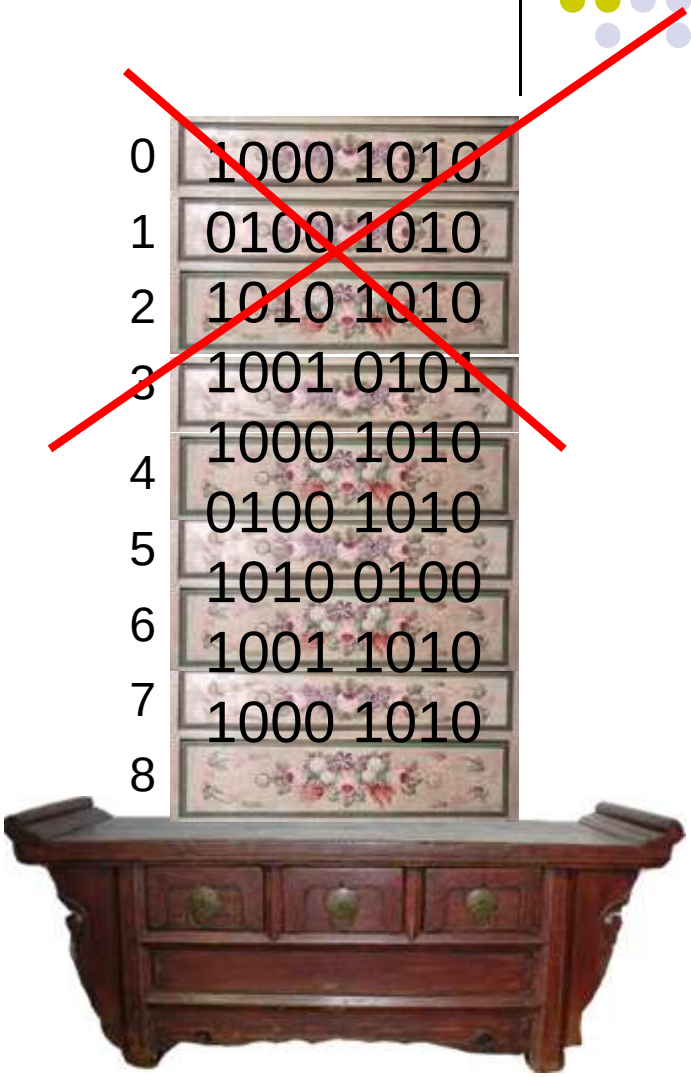
Ce sont les numéros de chaque tiroir

Afin de mieux s'y retrouver, on a fini par numéroter tous les tiroirs, et on leur attribue un numéro unique à chacun. C'est plus simple pour retrouver ce que l'on cherche n'est-ce pas? Alors on a commencé par le tiroir de tout en haut, qui porte le n° 0, puis 1 juste en dessous et ainsi de suite...



Zones sensibles

Tous les programmes sur un PC n'ont le droit d'utiliser qu'une certaine quantité de tiroirs qui leur est réservée, et le système d'exploitation nous garantit que personne d'autre ne va y toucher. De la même façon, si un programme essaie d'ouvrir un tiroir qui ne lui appartient pas, une erreur va être signalée.



Milions de tiroirs?

A nouveau, vu que tout ce qui est dans le pc est binaire, les numéros des tiroirs le sont aussi, et donc a chaque tiroir est associé une suite de 32 bits pour les représenter. En fait c'est tout simplement son numéro qui est converti en binaire.

Tiroir n° 0 = 0000 0000 0000 0000 0000 0000 0000 0000

Tiroir n° 1 = 0000 0000 0000 0000 0000 0000 0000 0001

Tiroir n° 2 = 0000 0000 0000 0000 0000 0000 0000 0010

Tiroir n° 3 = 0000 0000 0000 0000 0000 0000 0000 0011

Tiroir n° 4 = 0000 0000 0000 0000 0000 0000 0000 0100

Convention

Vu qu'il y a beaucoup de tiroirs je vais pas tous les dessiner à chaque fois, alors je mettrai juste ceux dont on a besoin.

Généralement on utilise plutôt la notation hexadécimale, c'est mieux adapté à la binarité de notre ami le pc. Alors, on rajoute un 0x devant l'adresse pour dire c'est pas en base 10, mais 16. Du genre, une adresse ça donne:

```
...
C2FF FFFF
C300 0000
C300 0001
C300 0003
C300 0004
...
```

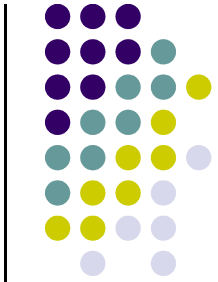
Le reste des tiroirs….

Le reste des tiroirs….

```
0x  03     EF    C1    20
   0000 0011 1110 1111 1100 0001 0010 0000
```

ça c'est les 32 bits de l'adresse du tiroir, en binaire, mais c'est moins cool.

Milions de tiroirs?



En langage d'informaticien, le fait d'écrire le numéro de chaque tiroir avec 32 bits se dit

« Les adresses sont codées sur 32 bits. »

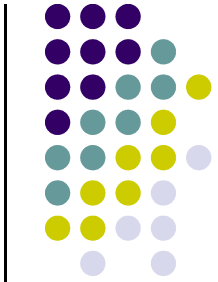
Cela signfie que on utilise une suite unique de 32 bits, unique dans le sens ou pour chaque 0101001… là, on a un seul tiroir qui lui est associé. Donc si on ne sait plus dans quel tiroir on a mis une information, on a une chance sur 2^{32} = 4'294'267'296 de trouver le bon tiroir du premier coup. En terme plus technique on dit que l'on dispose d'un espace d'adressage de 4 gigaoctets, ça fait toujours plus frime.

Pour information, t'as 500 fois meilleur temps de jouer au loto, puis là t'as une chance sur ''seulement'' 8'145'060.

Organisation de la mémoire

En réalité, les tiroirs sont mis sur 4 colonnes pour des raisons de stabilité, et éviter ainsi que la pile s'effondre. Ce qui veut dire que l'on peut stocker 32 bits (4 tiroirs à 8 bits chacun) par ligne. C'est pratique vu qu'il faut 4 tiroirs pour stocker un nombre entier ou à virgule! Par contre, un seul tiroir suffit pour mettre un caractère. Par la suite, je metterai les tiroirs sur une seule colonne afin qu'il me reste encore un peu de place sur les slides pour y mettre autre chose.

Taille des types



Chaque type de données nécessite un certain nombre de tiroirs pour être stockés, et cela peut varier selon le système d'exploitation utilisé. La fonction `sizeof(`*`type`*`)` retourne justement le nombre de tiroirs requis par variable. Voyons l'exemple suivant :

```
printf("%d",sizeof(char));    /*  prints 1 */
printf("%d",sizeof(short));   /*  prints 2 */
printf("%d",sizeof(int));     /*  prints 4 */
printf("%d",sizeof(long));    /*  prints 4 */
printf("%d",sizeof(float));   /*  prints 4 */
printf("%d",sizeof(double));  /*  prints 8 */
```

Organisation de la mémoire

Lorsque l'on déclare une variable dans un programme en C :

```
char monChar = 'e';
```

alors l'ordinateur nous donne à disposition un tiroir pour stocker ce caractère 'e'. Ce tiroir possède une adresse unique et la variable monChar est associée avec le contenu de ce tiroir.

Adresse	Contenu
C300 0000	01100101

Malheureusement, on ne peut mettre que des bits dans ce tiroir, et alors on doit trouver un moyen de coder le caractère 'e'. Pour cela, on a inventé le code ASCII qui associe le numéro 101 en décimal, 0x 65 en hexa et 0110 0101 en binaire.

Le tiroir qu'on nous donne n'est pas forcément vide.

Toujours initialiser les variables avant de les utiliser !!!

Je crois que je vais le mettre à la cave ce truc, on n'en a plus besoin pour l'instant

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

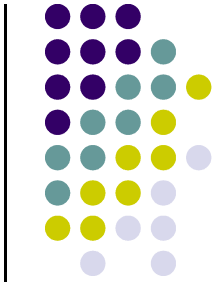
Source: www.asciitable.com

Extended ASCII Codes

128	Ç	144	É	161	í	177	▒	193	┴	209	╤	225	ß	241	±
129	ü	145	æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
130	é	146	Æ	163	ú	179	│	195	├	211	╙	227	π	243	≤
131	â	147	ô	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
132	ä	148	ö	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
133	à	149	ò	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
134	å	150	û	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
135	ç	151	ù	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
136	ê	152	ÿ	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
137	ë	153	Ö	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
138	è	154	Ü	171	½	187	╗	203	╦	219	█	235	δ	251	√
139	ï	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
140	î	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
141	ì	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
142	Ä	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	
143	Å	160	á	176	░	192	└	208	╨	224	α	240	≡		

Source: www.asciitable.com

Un pointeur?



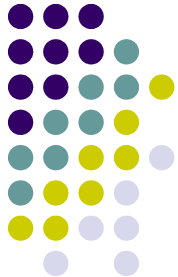
Un pointeur est un tiroir avec lequel on triche. Au lieu de mettre la donnée directement dans le tiroir, on va mettre l'adresse d'un autre tiroir, qui lui contiendra la donnée, à la place. C'est clair? Relis trois fois cette phrase et imprime-la dans ton coeur et tu verras comme tout te semblera plus simple.

Un pointeur se déclare ainsi : *type* nomVariable*

Exemples

```
int* largeur;
char* size;
long* foo;
```

Taille des types 2



Reprenons l'exemple avec les tailles des types, si maintenant on mets des pointeurs vers les mêmes types, cette fois on remarque que tous les pointeurs ont besoin de 4 tiroirs (32 bits par adrese), quels que soient les types vers lesquels ils pointent.

```
printf("%d",sizeof(char*));   /*  prints 4 */
printf("%d",sizeof(short*));  /*  prints 4 */
printf("%d",sizeof(int*));    /*  prints 4 */
printf("%d",sizeof(long*));   /*  prints 4 */
printf("%d",sizeof(float*));  /*  prints 4 */
printf("%d",sizeof(double*)); /*  prints 4 */
```

Mon premier pointeur

N'oubliez pas qu'en réalité on y écrit 32 bits, je le mets en hexa par simplicité

Déclarons un tiroir lettre qui contient le caractère 'e':

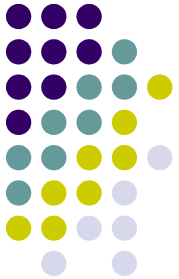
```
char lettre = 'e';
```

Disons que l'ordinateur va mettre le 'e' dans le tiroir n° **DF00 4301.** Ensuite on va déclarer un pointeur vers le tiroir dans lequel on a mis le 'e' (c'est à dire que le pointeur devra contenir dans son tiroir l'adresse du tiroir ou est le 'e').

```
char* pointeur = &lettre;
```

Pour cela, j'ai du mettre un **&** devant la variable, car c'est un opérateur qui retourne l'adresse du tiroir associé à une variable.

&, * et :-)



Mettre l'***** devant une variable oblige le compilateur à interpréter le contenu du tiroir de la variable comme un pointeur (donc l'addresse d'un autre tiroir!!!) et va ainsi accéder au contenu du tiroir situé à l'adresse du tiroir 1. Cela ne retourne pas le contenu du tiroir directement associé à la variable!!!!

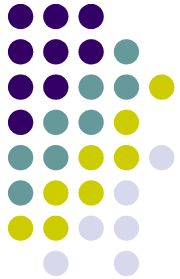
Donc quand vous tapez **`*lettre`**, vous êtes en train de dire à l'ordi:
« Accède au tiroir dont l'adresse se trouve dans le tiroir associé à **`lettre`**»

Donc, attention de ne PAS mettre l'* devant une variable dont le tiroir ne contient pas l'adresse valide d'un autre tiroir du système.

Donc, après avoir déclaré **`int lettre = 10;`** la commande **`*lettre = 15`**, va être comprise ainsi : « Vas mettre 15 dans le tiroir dont l'adresse se trouve contenue dans le tiroir **`lettre`**»

C'est à dire mets 15 dans le tiroir qui se trouve à l'adresse 10.
!!! A éviter, car on ne sait pas ce qu'il y a dans le tiroir n° 10 !!!

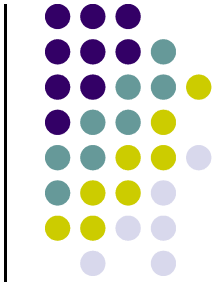
Pas compris



Quand on dit que la variable **`monPointeur`** est un pointeur vers un entier, alors on est en train de dire que dans le tiroir associé à la variable **`monPointeur`** on va trouver l'addresse d'un autre tiroir, qui lui contient l'entier (pointé par **`monPointeur`**, n'est-ce pas!).

C'est clair?

C’est pas clair ok.



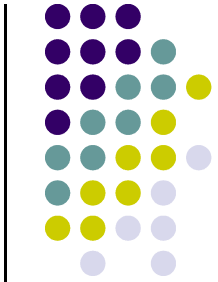
Un pointeur vers un entier est l’adresse d’un tiroir qui contient un entier.

Un pointeur vers un pointeur vers un entier est l’adresse d’un tiroir qui contient l’adresse d’un AUTRE tiroir qui lui contient un entier…

Et ainsi de suite…

C’est bon cette fois?

Toujours pas?



Non? Bon.

Un pointeur est une adresse.

Un pointeur vers un pointeur est un adresse à laquelle on trouve une autre adresse.

Chaînes de caractères

Vu que le type String n'existe pas en C, alors on le simule en mettant les caractères dans une suite consécutive de tiroirs, à raison d'un caractère par tiroir.

Pour que ce soit bien fait, il faut obligatoirement mettre un le caractère '\0', dans le dernier tiroir afin de savoir où se termine la chaîne.

Donc printf("%s",lettre3), affiche tous les caractères qu'il trouve jusqu'au prochain \0.

```
Char* lettre3 = "Salut";
```

Interpretation

Là, ça va devenir intéressant :

Il est impératif de savoir ce que l'on fait en programmant!

Vu que il n'y a que des bits dans les tiroirs, rien ne nous indique à quoi ils correspondent, c'est-à-dire savoir s'il s'agit d'un entier, d'un caractère, d'une adresse… Rien n'indique non plus comment on doit les utilliser. De ce fait, il est essentiel d'être certain que l'on interprète de la bonne façon le contenu d'un tiroir. Voyons un petit exemple très amusant :

on déclare :

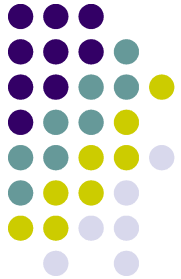
```
char lettre2 = 'c';
```

Je vous rappelle que c'est le code ASCII du caractère 'c' en binaire.

On ne sait pas ce que c'est,
Alors ATTENTION!!!

Trop classe, ça fait tellement ambassadeur!

Interpretation



Étant donné que le C est un langage puissant, nous avons le droit d'interpréter la variable lettre2 comme n'importe quoi, mais cela donnera généralement un résultat incohérent. La preuve :

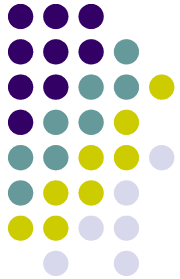
Si on fait `printf("%c",lettre2);`

Ben ça affiche le caractère 'c', nickel!

Mais si on fait `printf("%s",lettre2);`

Ben ça va pas marcher car il va lire tous les tiroirs après celui de lettre2 jusqu'à ce qu'il trouve un \0, mais rien ne nous garantit qu'il va en trouver un. D'où de fortes chances de se retrouver avec un segmentation fault ou un bus error (c'est le genre d'erreurs qui arrivent quand on essaie d'accéder à des tiroirs qui nous appartiennent pas, comme je disais avant).

malloc et autres créatures



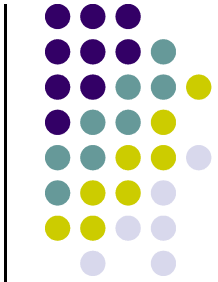
Si on a besoin d'un certain nombre de tiroirs en cours d'exécution du programme (par exemple, on doit créer un tableau dont la taille est entrée par l'utilisateur), on ne peut pas déclarer le tableau à l'avance vu qu'on ne connaît pas encore sa taille!

Alors on doit pouvoir allouer le nombre de tiroirs requis en cours d'exécution du programme, alors la commande **`malloc`** de la librairie **`stdlib.h`** (n'oubliez donc pas de mettre **`#include <stdlib.h>`** au début de votre code) nous permet de nous réserver la quantité de mémoire(de tiroirs!) voulue, disons N. Cettre commande retourne une addresse (donc un pointeur) vers le premier tiroir de cette zone de N tiroirs. Elle fonctionne ainsi **`malloc(nbre_de_tiroirs).`** Comme c'est facile, non? Mais on doit encore préciser le type du pointeur, c'est-à-dire le convertir explicitement (typecast) comme étant du type du contenu pointé.

Exemple : on veut 4 tiroirs pour y mettre "baba" (1 tiroir peut contenir 1 seul caractère), donc il nous faut 4 tiroirs :

```
char* monPointeur = (char*) malloc(4*sizeof(char));
monPointeur = "baba";
```

Test final



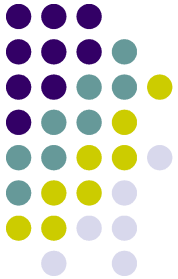
Vous croyez avoir tout compris? C'est ce que l'on va voir! Pour cela, vous devez passer le petit test sur les slides suivants et deviner à quoi correspond la myriade de déclarations en C:

Exemple

Question : C'est quoi un **`int* f()`**?

Réponse : C'est une fonction qui retourne un pointeur sur un entier.

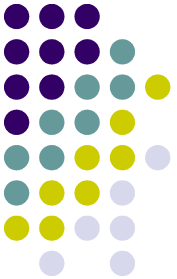
Test final : Round 1



`int i`	un entier
`int* p`	un pointeur sur un entier
`int** q`	un pointeur sur un pointeur sur un entier
`int* f()`	une fonction qui retourne un pointeur sur un entier

Ces exemples ont été repris du cours de C du Prof J.-C. Chappelier.

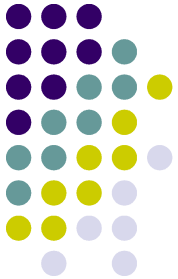
Test final : Round 2



`int (*f)()`	un pointeur sur une fonction qui retourne un entier
`int* (*f)()`	un pointeur sur une fonction qui retourne un pointeur sur un entier
`int** f()`	une fonction qui retourne un pointeur sur un pointeur sur un entier
`int* t[]`	un tableau de pointeurs sur des entiers
`int (*p)[]`	un pointeur sur un tableau d'entiers

Ces exemples ont été repris du cours de C du Prof J.-C. Chappelier.

Test final : Round 3



`int (*f())[]` une fonction qui retourne un pointeur sur un tableau d'entiers

`int* (*f())()` une fonction qui retourne un pointeur sur une fonction retournant un pointeur sur un entier

Ces exemples ont été repris du cours de C du Prof J.-C. Chappelier.

Final Round

```
int (*(*(*f())[])())[]
```

une fonction qui retourne un pointeur sur un tableau de pointeurs pointant sur des fonctions retournant des pointeurs sur un tableau d'entiers

Ces exemples ont été repris du cours de C du Prof J.-C. Chappelier.

Conclusion

Voilà, c’est déjà fini et sans le savoir vous venez de comprendre[1] l’un des concepts qui est la cause de la calvitie de la plupart des informaticiens, et ceci sans vous arracher le moindre cheveu.

Bravo.

[1] Si vous n’avez pas encore compris, je vais bientôt adapter ce document pour en faire une version adaptée aux enfants entre 3 et 6 ans. A paraître.