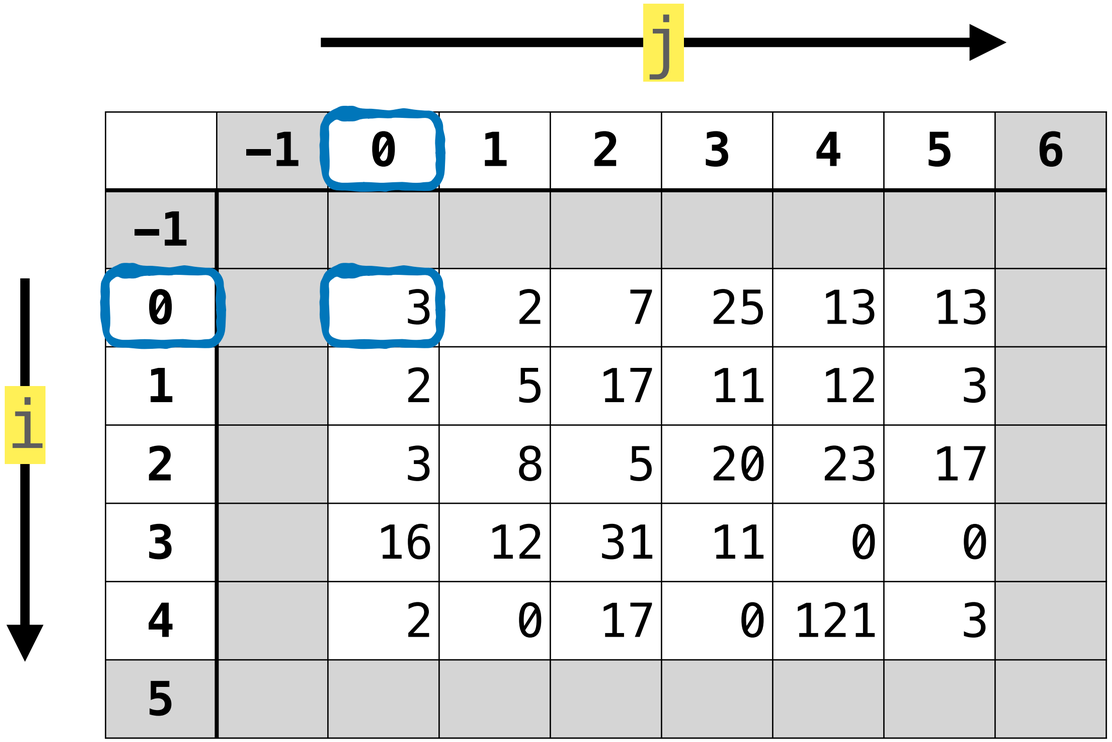
Pointeurs

et la mémoire

ICC-C Cours 6

Parcourir un tableau 2-d

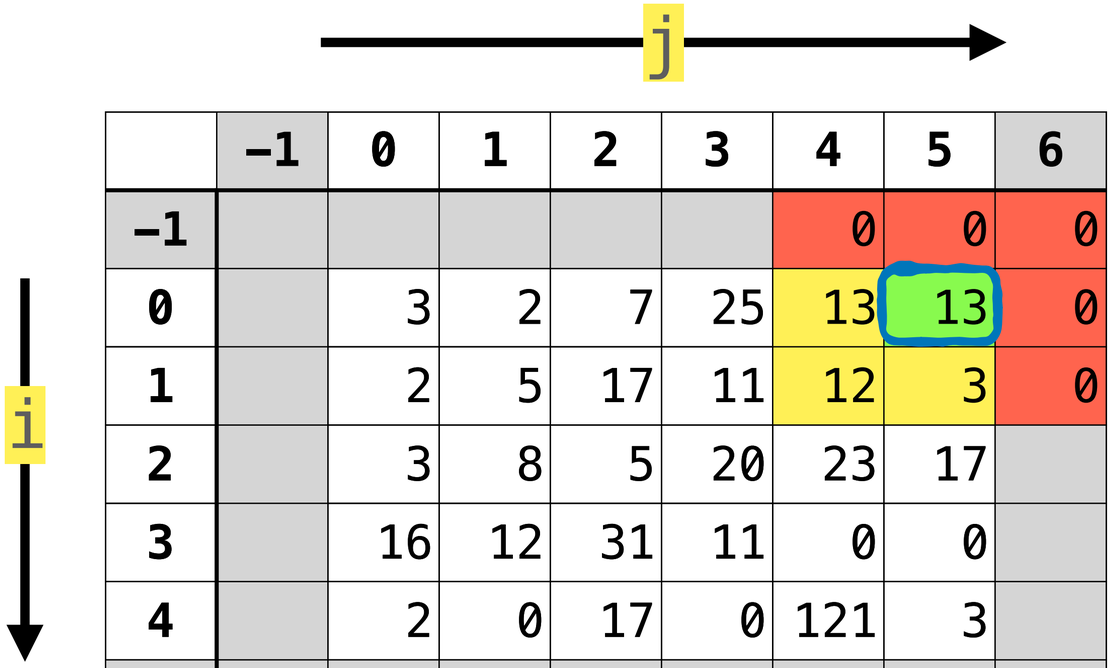


	-1	0	1	2	3	4	5	6
-1								
0		3	2	7	25	13	13	
1		2	5	17	11	12	3	
2		3	8	5	20	23	17	
3		16	12	31	11	0	0	
4		2	0	17	0	121	3	
5								

etc.

```
for (i = 0; i < 5; i++)
{
    for (j = 0; j < 6; j++)
    {
        printf("%d ", map[i][j]);
    }
    printf("\n");
}
```

Indices valides 2-d

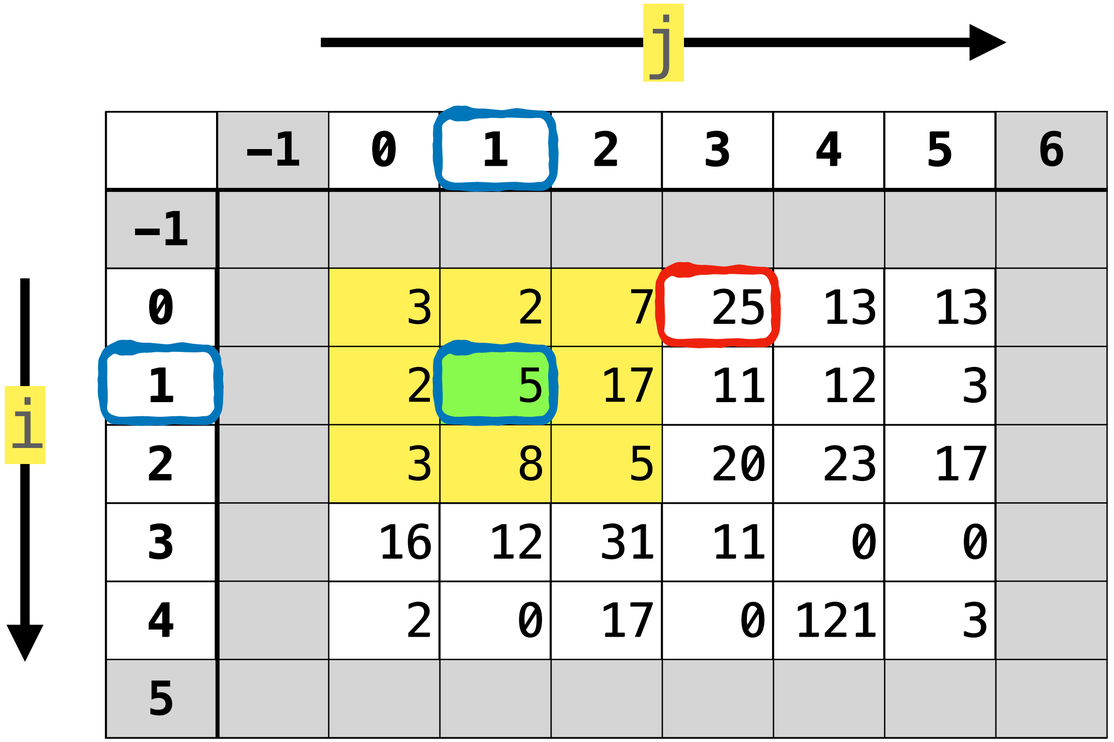


	-1	0	1	2	3	4	5	6
-1						0	0	0
0		3	2	7	25	13	13	0
1		2	5	17	11	12	3	0
2		3	8	5	20	23	17	
3		16	12	31	11	0	0	
4		2	0	17	0	121	3	
5								

```
value_or_zero(map, 0, 5) -> 13
value_or_zero(map, -1, 6) -> 0
```

```
int value_or_zero(
    int map[100][100], int i, int j
)
{
    if (i < 0 || i >= M ||
        j < 0 || j >= N)
    {
        return 0;
    }

    return map[i][j];
}
```

Calculer le max



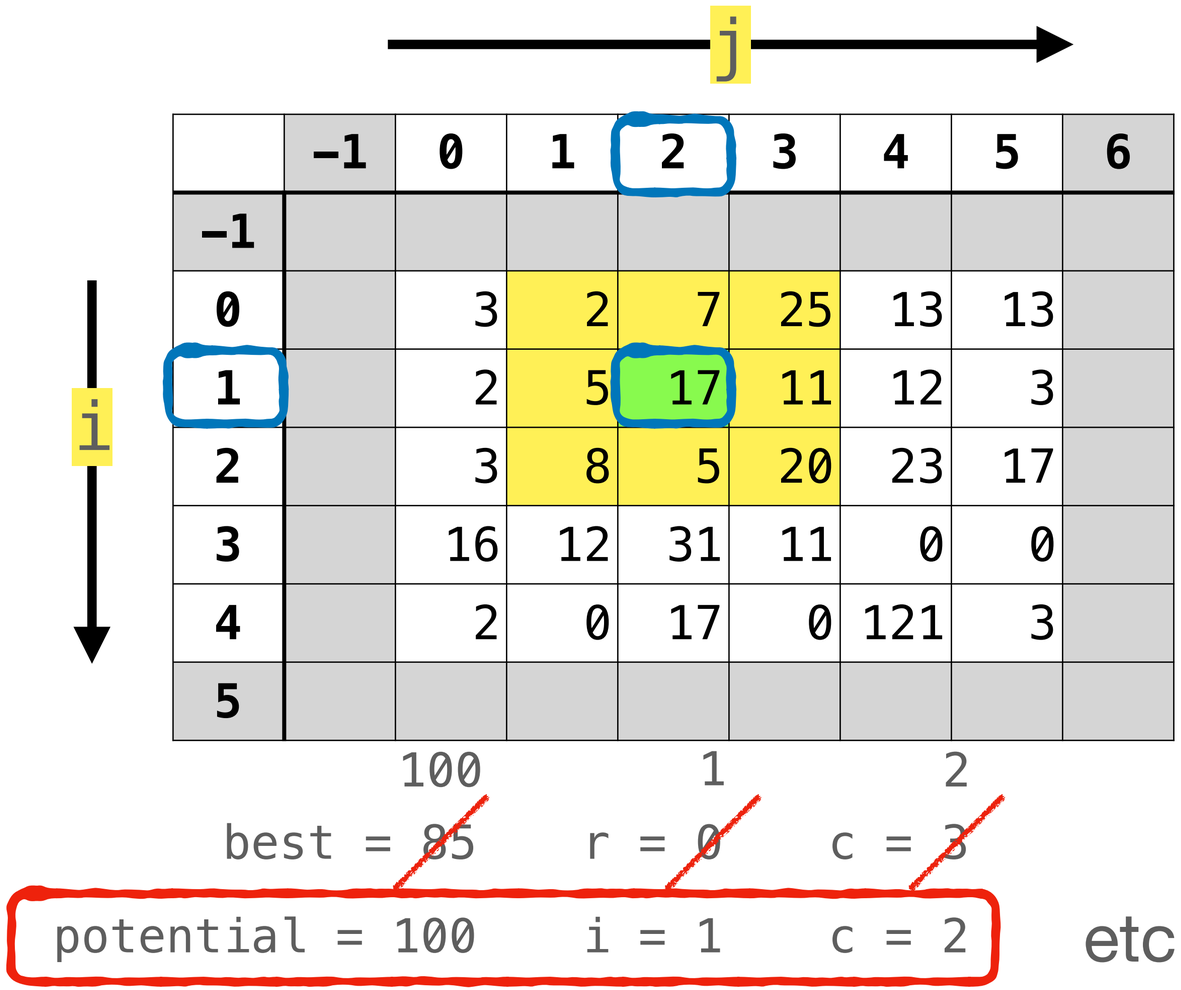
	-1	0	1	2	3	4	5	6
-1								
0		3	2	7	25	13	13	
1		2	5	17	11	12	3	
2		3	8	5	20	23	17	
3		16	12	31	11	0	0	
4		2	0	17	0	121	3	
5								

best = 85 r = 0 c = 3

potential = 52 i = 1 c = 1

```
int best = 0, r, c;

for (i = 0; i < M; i++)
{
    for (j = 0; j < N; j++)
    {
        int potential = estimate(i, j);

        if (potential > best)
        {
            best = potential;
            r = i;
            c = j;
        }
    }
}
```

Calculer le max




```
int best = 0, r, c;

for (i = 0; i < M; i++)
{
    for (j = 0; j < N; j++)
    {
        int potential = estimate(i, j);

        if (potential > best)
        {
            best = potential;
            r = i;
            c = j;
        }
    }
}
```

La mémoire


Don't you have ANYTHING USEFUL in there?!
Pretty much just song lyrics and movie quotes.
MEMORIES
theAwkwardYeti.com

Qu’est-ce qu’un caractère ?

```c
printf("Caractère '%c'\n", 'A');
// Affiche : Caractère 'A'

printf("Caractère %d\n", 'A');
// Affiche : Caractère 65

printf("Caractère %d\n", '\n');
// Affiche : Caractère 10
```

- Chaque caractère est en fait un entier 😲
- C’est le code du caractère qu’on va effectivement afficher

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[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	[NEGATIVE 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 OF 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	[SUBSTITUTE]	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	[DEL]

Une somme

```c
char x = 12;
char y = 12;
char z = x + y;

printf("%d + %d = %d\n", x, y, z);

// Affiche: 12 + 12 = 24
```

Une autre somme

```c
char x = 120;
char y = 120;
char z = x + y;

printf("%d + %d = %d\n", x, y, z);

// Affiche: 120 + 120 = -16
```

Limites physiques

⚠️

- Débordement (*overflow*)
- Un entier occupe une **place limitée** dans la mémoire
- `char` occupe 1 octet (*1 byte*) = 8 bit
  - donc de -128 à 127

```
*  64 32 16  8  4  2  1
0  0  1  1  0  0  0  0  = 16 + 32 = 48
```

Pareil pour les `int`

```c
int big_a = 1500000000;
int big_b = 1500000000;

printf("%d + %d = %d\n”, big_a, big_b, big_a + big_b);

// Affiche: 1500000000 + 1500000000 = -1294967296
```

- On peut aussi faire déborder les `int`
- Un `int` occupe en général 4 octets (suivant le compilateur…)

Types et leurs tailles

Taille en bytes	1 byte	2 bytes	4 bytes	8 bytes
Entiers	`char`	`short`	`int`	`long`
Rationnels			`float`	`double`
Caractères	`char`			
Rien	`void`			

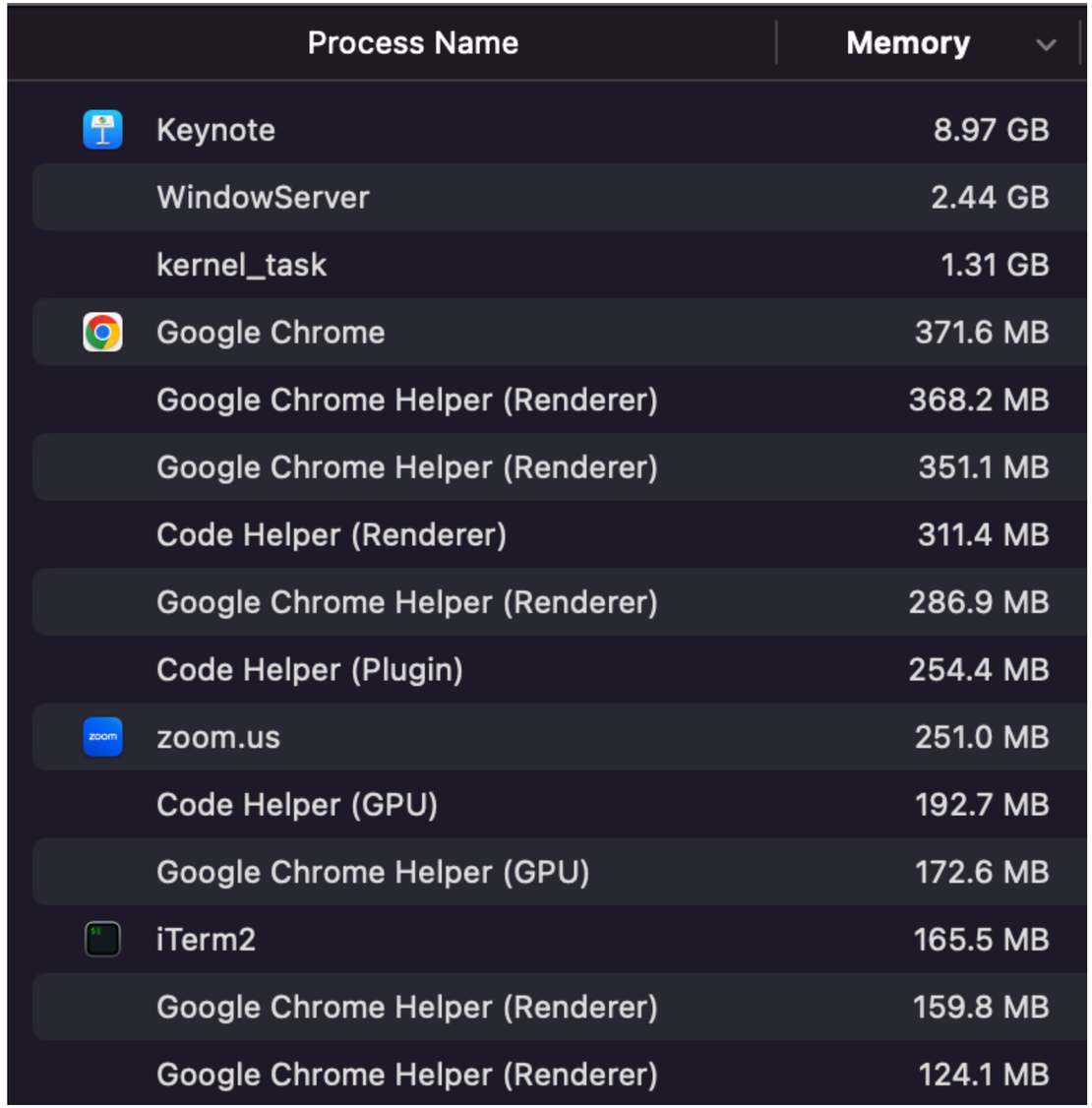
sizeof

- Si on veut savoir combien de place prend un certain type, il faut utiliser l'opérateur `sizeof`
- "Retourne" un `long` / format `"%ld"`
- Aussi pour des valeurs et des variables
- Même pour des tableaux
  - …car `3 x sizeof(int) = 12`
- Et des strings

```c
printf("Le type int occupe %ld bytes\n",
       sizeof(int));
// Affiche:
// Le type int occupe 4 bytes

int tableau[] = {1, 2, 3};
printf("%ld\n", sizeof(tableau));
// Affiche: 12

printf("\"Hello\" occupe %ld bytes\n",
       sizeof("Hello"));
// Affiche: "Hello": occupe 6 bytes
```

La mémoire

- Une (loooongue) suite d’octets (*bytes*)
- 32GB RAM = 34`359`738`368 bytes
- Une partie est réservée pour le système d’exploitation
- La plupart va aux programmes de l’utilisateur

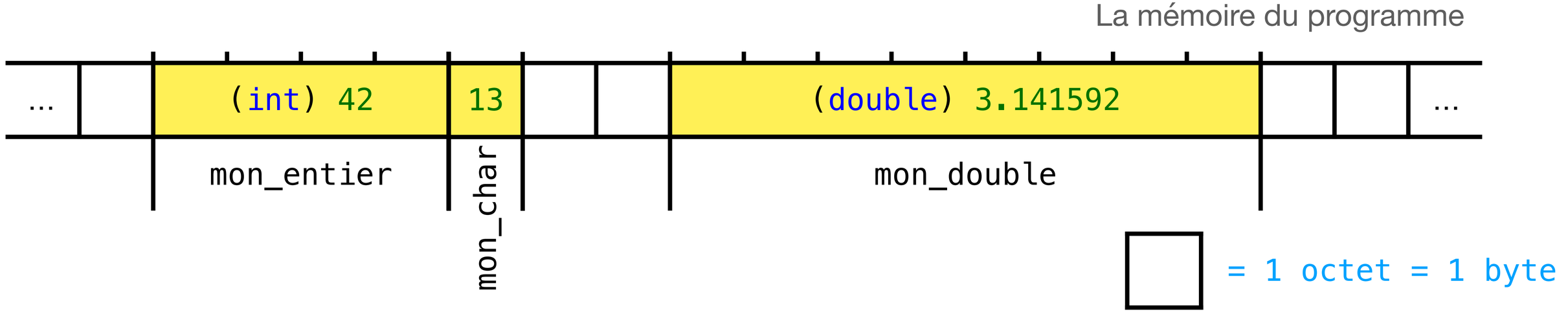


Process Name	Memory
Keynote	8.97 GB
WindowServer	2.44 GB
kernel_task	1.31 GB
Google Chrome	371.6 MB
Google Chrome Helper (Renderer)	368.2 MB
Google Chrome Helper (Renderer)	351.1 MB
Code Helper (Renderer)	311.4 MB
Google Chrome Helper (Renderer)	286.9 MB
Code Helper (Plugin)	254.4 MB
zoom.us	251.0 MB
Code Helper (GPU)	192.7 MB
Google Chrome Helper (GPU)	172.6 MB
iTerm2	165.5 MB
Google Chrome Helper (Renderer)	159.8 MB
Google Chrome Helper (Renderer)	124.1 MB

Les variables

- Le compilateur associe une zone de mémoire contiguë à chaque variable
- Cette zone est **figée** tant que la variable est “définie”

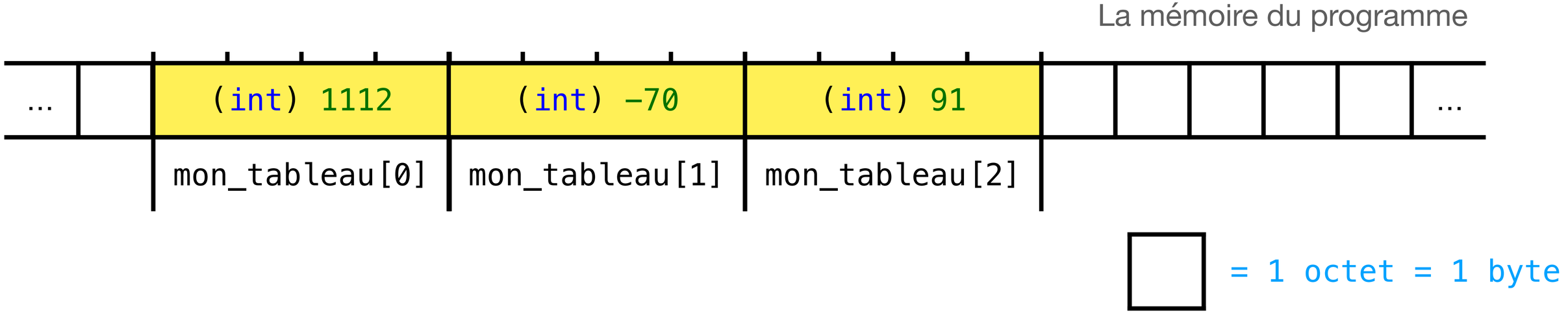
```
int mon_entier = 42;
char mon_char = 13;
double mon_double = 3.141592;
```



Les tableaux

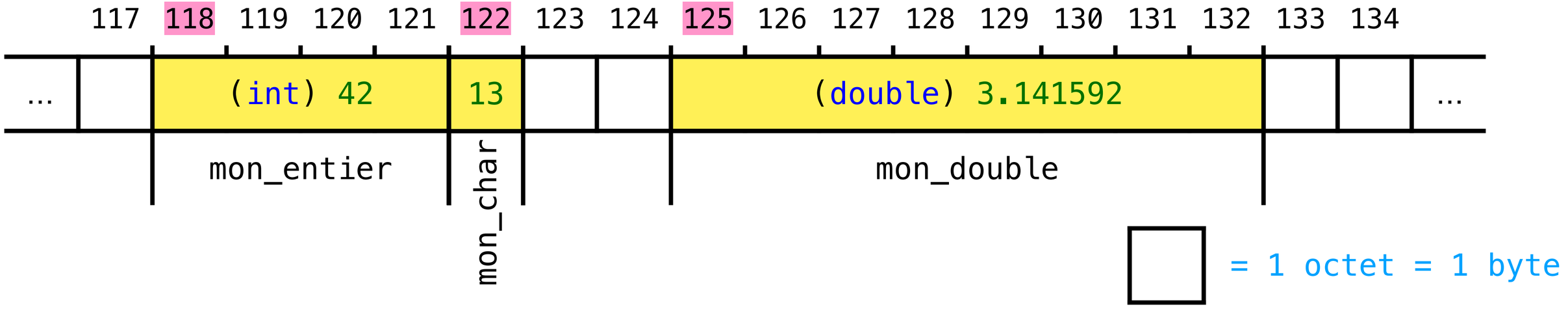
```
int mon_tableau[3] = {1112, -70, 91};
```

`sizeof(mon_tableau)` est 12



Adresses mémoire

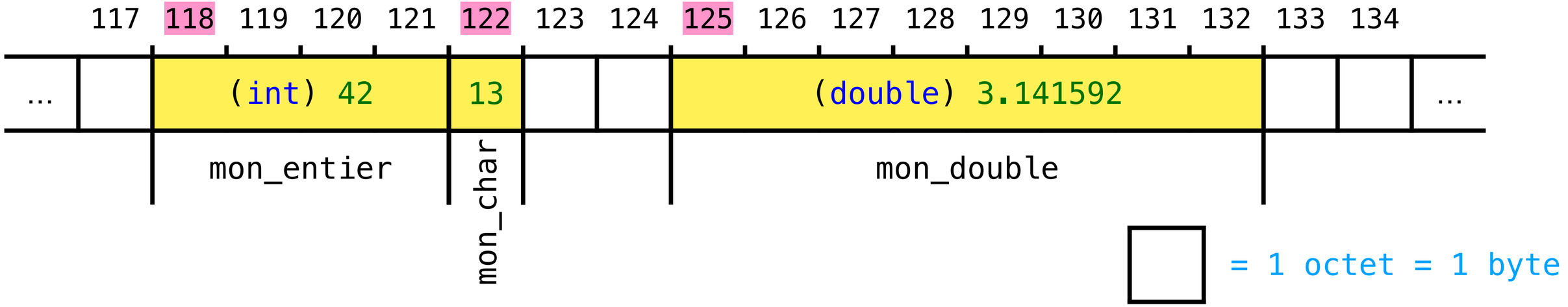
- Chaque octet dans la mémoire du programme a une adresse
- C'est comme l'indice dans un énorme tableau d'octets
- L'adresse d'une variable = l'adresse du premier octet qu'elle occupe
  - "La variable `mon_entier` se trouve à l'adresse 118", etc.



L’opérateur &

“Adresse de” (*Address of*)

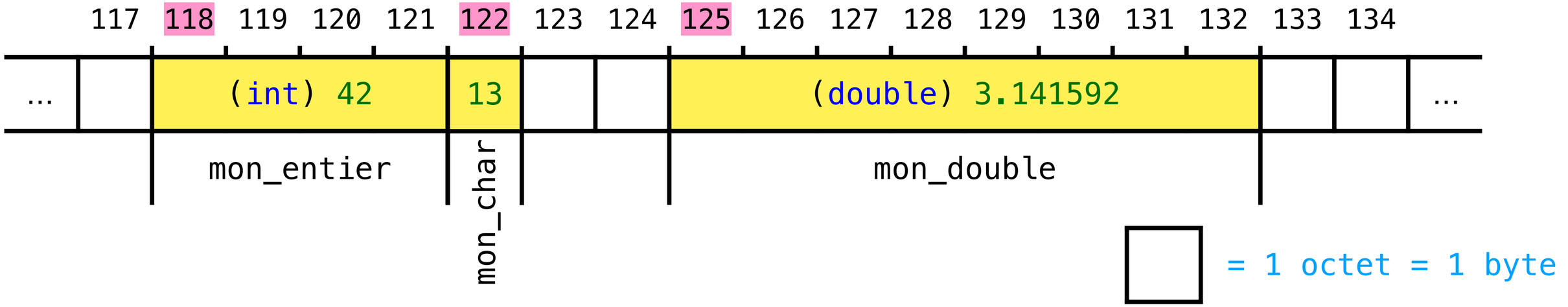
- Permet d’obtenir l’adresse d’une variable/L-valeur
- Quand on écrit
  `&mon_entier`
  `&mon_char`
  `&mon_double`
  on obtient l’adresse de ces variables



Les pointeurs

- `&mon_entier` a un type spécial pour indiquer que c'est une adresse
- C'est un pointeur vers un `int` (= `int` *pointer*)

```
int *ptr = &mon_entier; // une variable qui contient
                        // une adresse vers un int
```



Les pointeurs

- Un pointeur vers un certain type `T` aura le type `T*`
- ⚠️ On peut déclarer des variables de type “T” et “pointeur vers T” dans la même instruction

```
int entier, *ptr, tableau[10];
```

Variable entière

Pointeur vers entier

Tableau de 10 entiers

La taille d’un pointeur

- Une variable pointeur stocke une adresse de mémoire
- Elle doit elle-même être stockée en mémoire 🤯

```
printf("La taille d'un pointeur est %ld octets\n",
        sizeof(ptr));
// Affiche: La taille d'un pointeur est 8 octets
```

- Un pointeur peut aller de 0 à 18`446`744`073`709`551`615

L’opérateur *

Opérateur d’indirection (*dereference operator*)

- Permet d’obtenir la valeur qui se trouve à l’adresse stockée dans le pointeur

```c
int mon_entier = 42;
int *ptr = &mon_entier;

printf("L'adresse de ma variable est %p\n", ptr);
// Affiche: L'adresse de ma variable est 0x7ffeeb0b4a30

printf("La valeur de mon_entier est %d\n”, *ptr);
// Affiche: La valeur de mon_entier est 42
```

Valeur hexadécimale = nombre en **base 16** (hex pour les intimes)

L’opérateur *

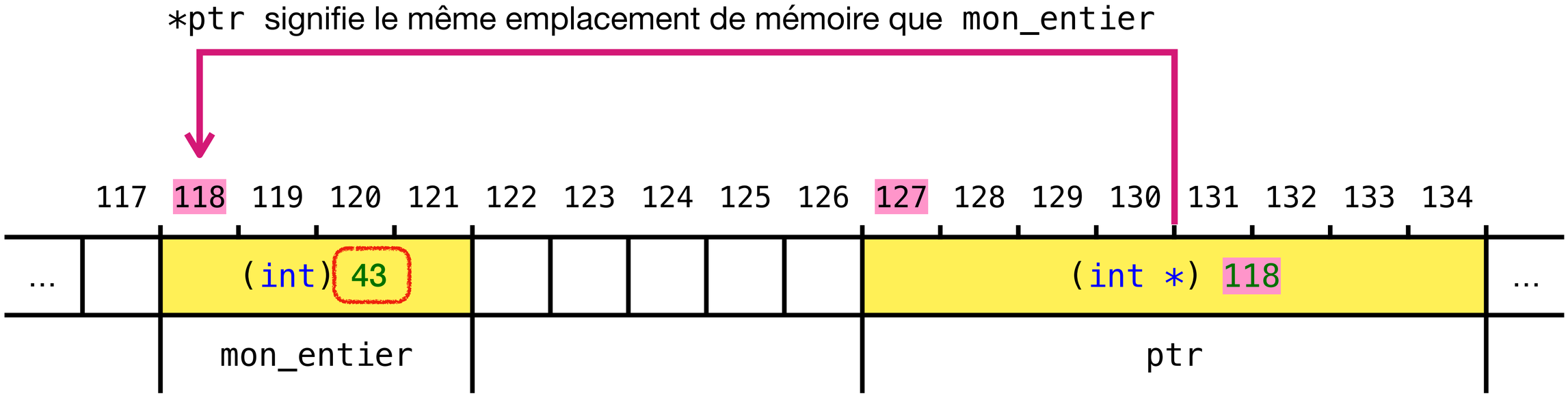
Opérateur d’indirection (*dereference operator*)

- En fait on obtient une L-valeur!

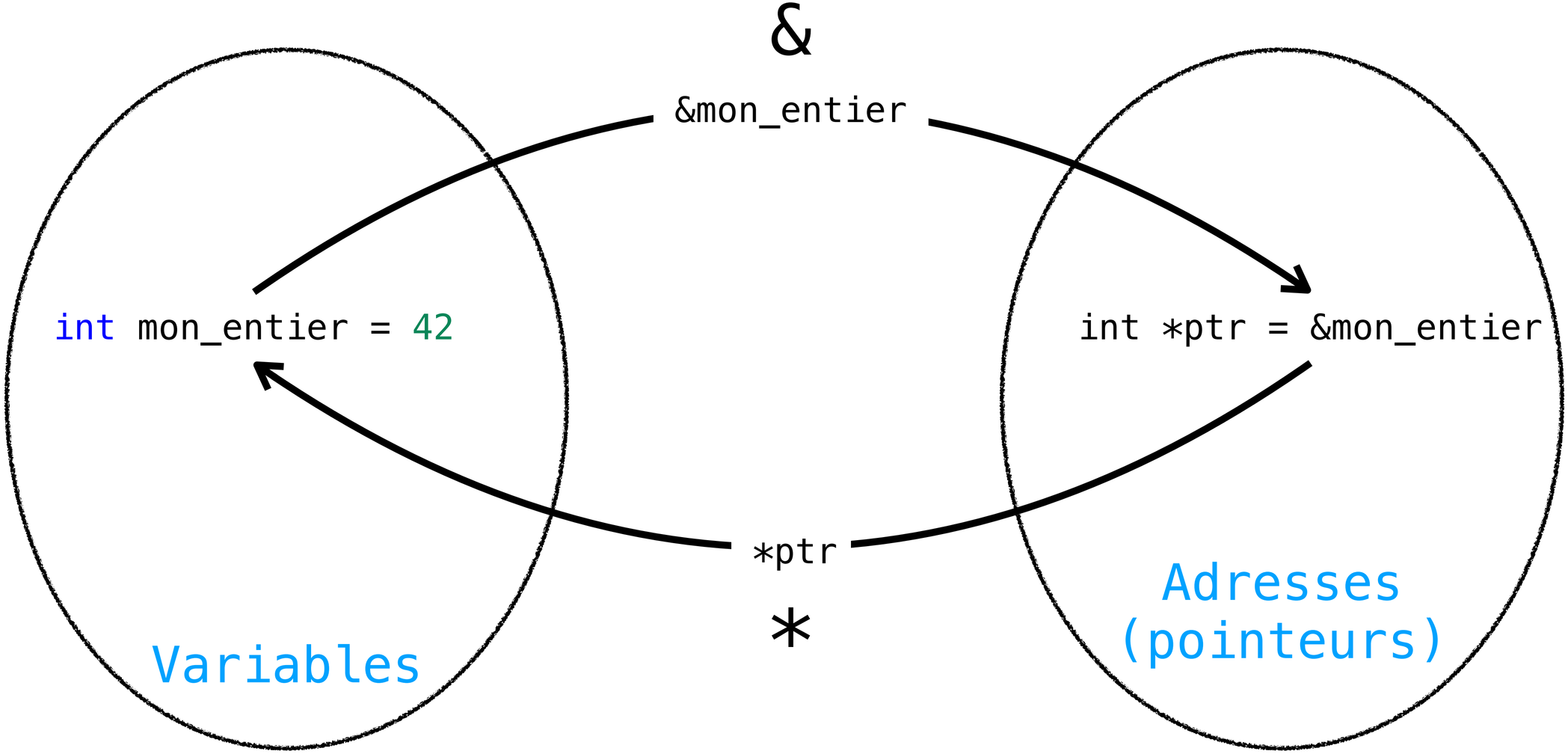
```
int mon_entier = 42;
int *ptr = &mon_entier;

*ptr = 43;

printf("La valeur de mon_entier est %d\n",
        mon_entier);
// Affiche: La valeur de mon_entier est 43
```

Que s’est-il passé ?

```c
int mon_entier = 42;
int *ptr = &mon_entier;
*ptr = 43;
printf("La valeur de mon_entier est %d\n",
       mon_entier);
// Affiche: La valeur de mon_entier est 43
```



Résumé


&
&mon_entier
int mon_entier = 42
int *ptr = &mon_entier
*ptr
*
Variables
Adresses
(pointeurs)

Les pointeurs - mode d'emploi

- Souvenez-vous
- La fonction `incrementer` ne change pas la variable `a`
- Affiche:

```
main avant: a = 1
incrémenter: a = 2
main après: a = 1
main b = 2
```

```c
int incrementer(int a)
{
  a = a + 1;
  printf("incrémenter: a = %d\n", a);
  return a;
}

int main()
{
  int a = 1, b = 0;
  printf("main avant: a = %d\n", a);

  b = incrementer(a);

  printf("main après: a = %d\n", a);
  printf("main b = %d\n", b);
}
```

Les pointeurs - mode d'emploi

- Nouvelle fonction `incrementer2` qui permet de changer la variable a
- Devrait afficher:

```
main avant: a = 1
incrémenter2: a = 2
main après: a = 2
main b = 2
```

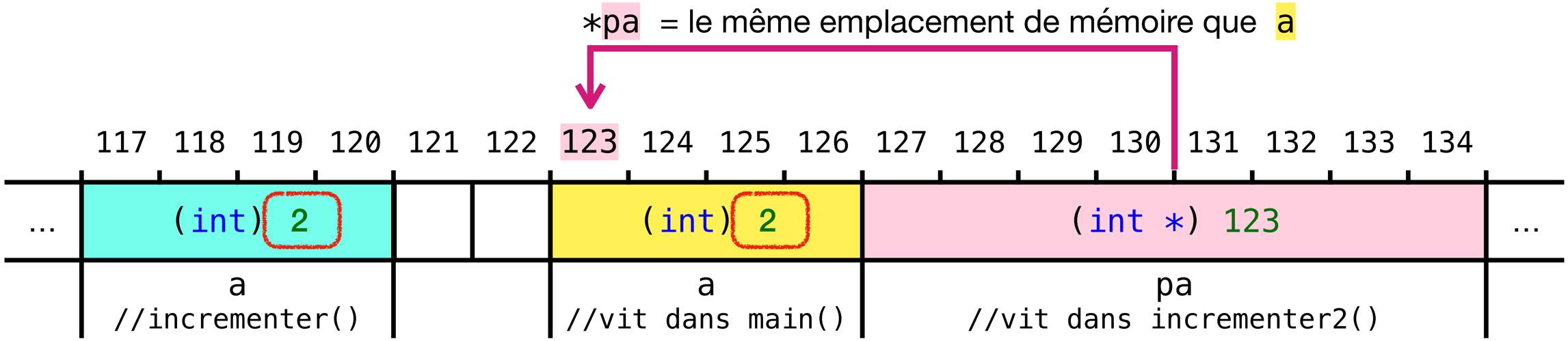
- Différences:
  1. Le paramètre est un pointeur
  2. (Conséquence) On appelle avec l'adresse de a

```c
int incrementer2(int *pa);

int main()
{
  int a = 1, b = 0;
  printf("main avant: a = %d\n", a);

  b = incrementer2(&a);

  printf("main après: a = %d\n", a);
  printf("main b = %d\n", b);
}
```

Les pointeurs - mode d'emploi

```c
int incrementer(int a)
{
  a = a + 1;
  printf("incrémenter: a = %d\n", a);
  return a;
}
```

```c
int incrementer2(int *pa)
{
  *pa = *pa + 1;
  printf("incrémenter2: a = %d\n", *pa);
  return *pa;
}
```



Conclusion

- Les pointeurs sont aussi passés par valeur
- Leur contenu par contre permet de retrouver l'emplacement de mémoire qui nous intéresse
- Le mystère du & dans les appels à `scanf` est résolu
- Et les tableaux alors ?