Les pointeurs

Manipulation d'adresses et de ce qui est contenu dans ces adresses

Très important, fondamental même en C

mauvaise réputation : 'dur à comprendre', 'difficile à utiliser', 'écriture impossible à interpréter'

comprendre ce qu'est une adresse $\rightarrow$ emploi quasi naturel des pointeurs et des notations associées

Les pointeurs sont un outil puissant !

Les adresses

Retour sur les variables :

4 caractéristiques :

- un **type**
- un **nom**
- une **valeur**
- une **adresse**

les 3 premières déjà bien exploitées. Mais la 4ème ?

Les adresses

Schématisation par une boîte ou cellule :

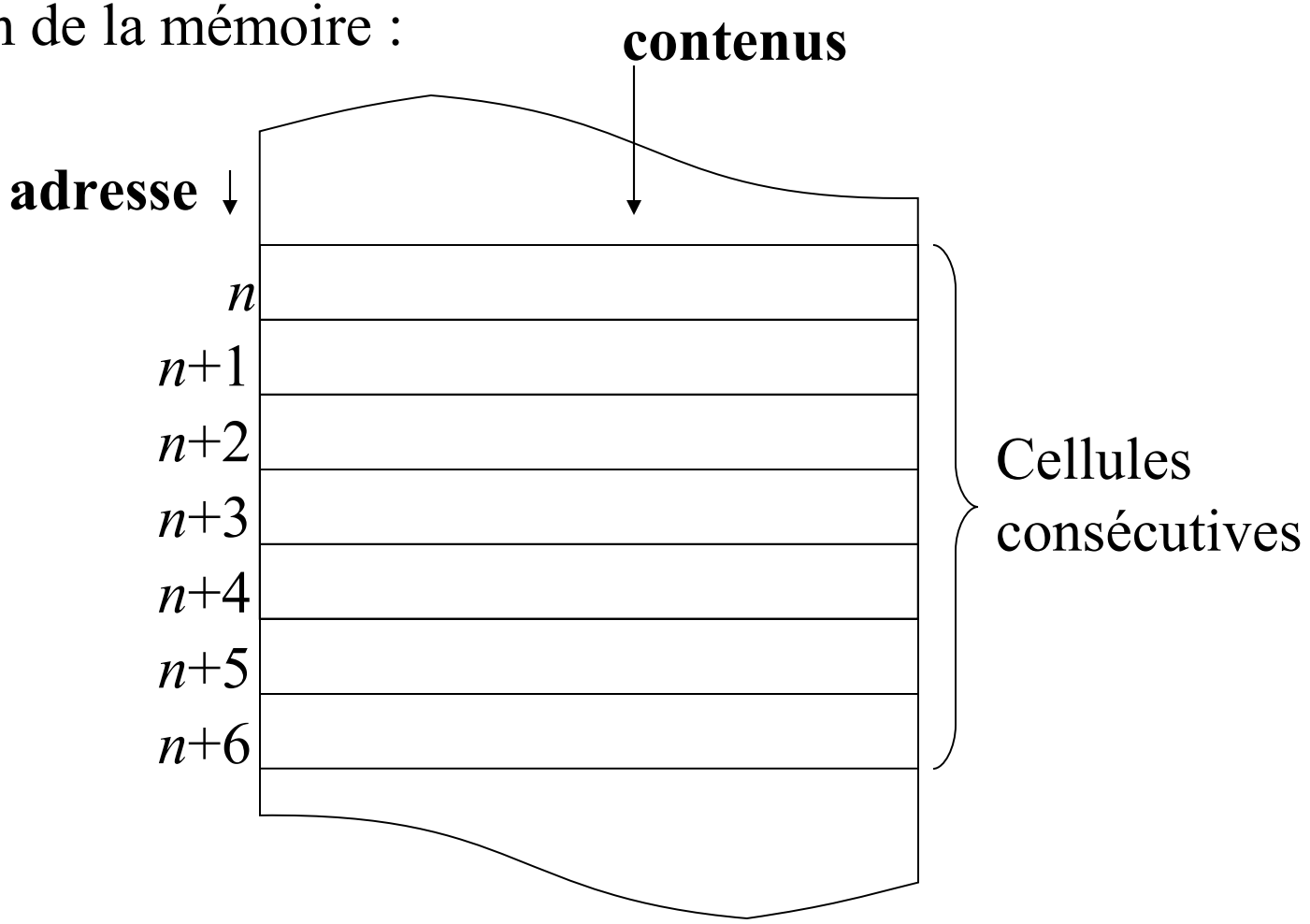
l'ordinateur doit stocker cette variable quelque part : dans la RAM

les octets (ou cellules) de la RAM sont numérotés pour que la machine s'y retrouve : chaque cellule à une **adresse**, et peut contenir une **valeur** (son **contenu**)

Attention à la distinction **adresse**/**contenu**

Les adresses

Schématisation de la mémoire :



Les adresses

Les adresses démarrent à 0.

Lors de la déclaration de variable, le compilateur associe automatiquement une adresse à la variable → inconnue du programmeur.

Le programmeur connaît juste le nom de la variable.

a chaque fois que le compilateur rencontre le nom : il en déduit l'adresse.

Possède une table de correspondance : nom de variable ⇔ adresse de cette variable en mémoire.

Les adresses

Illustration par un petit programme

```
void main()
{
        int val1,val2;

        val1 = 3;
        val2 = -2*val1+4;

        if (val2 < val1)
                {
                printf("%d < %d\n",val2,val1);
                }
}
```

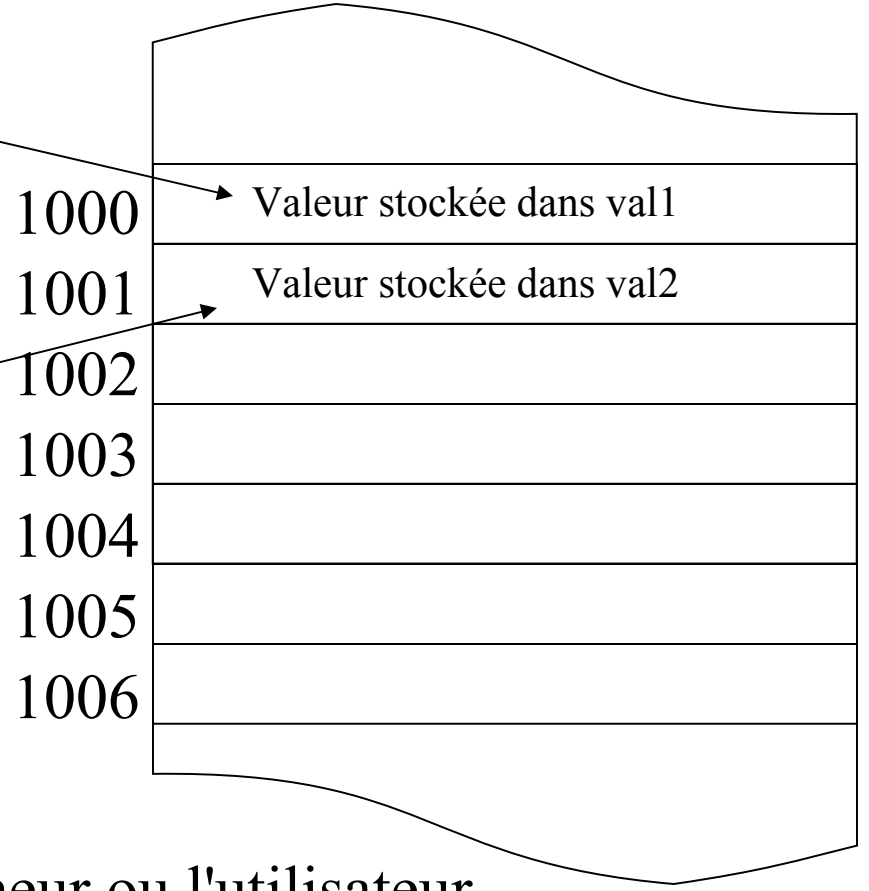
Le compilateur utilise des cellules en mémoire pour stocker ces valeurs. Il les 'alloue'.

Exemple : `val1` à l'adresse 1000, `val2` à l'adresse 1001 (arbitraire)

Les adresses

Lorsque l'on utilise val1, la machine va consulter l'adresse 1000

Lorsque l'on utilise val2, la machine va consulter l'adresse 1001



Invisible pour le programmeur ou l'utilisateur

Les adresses

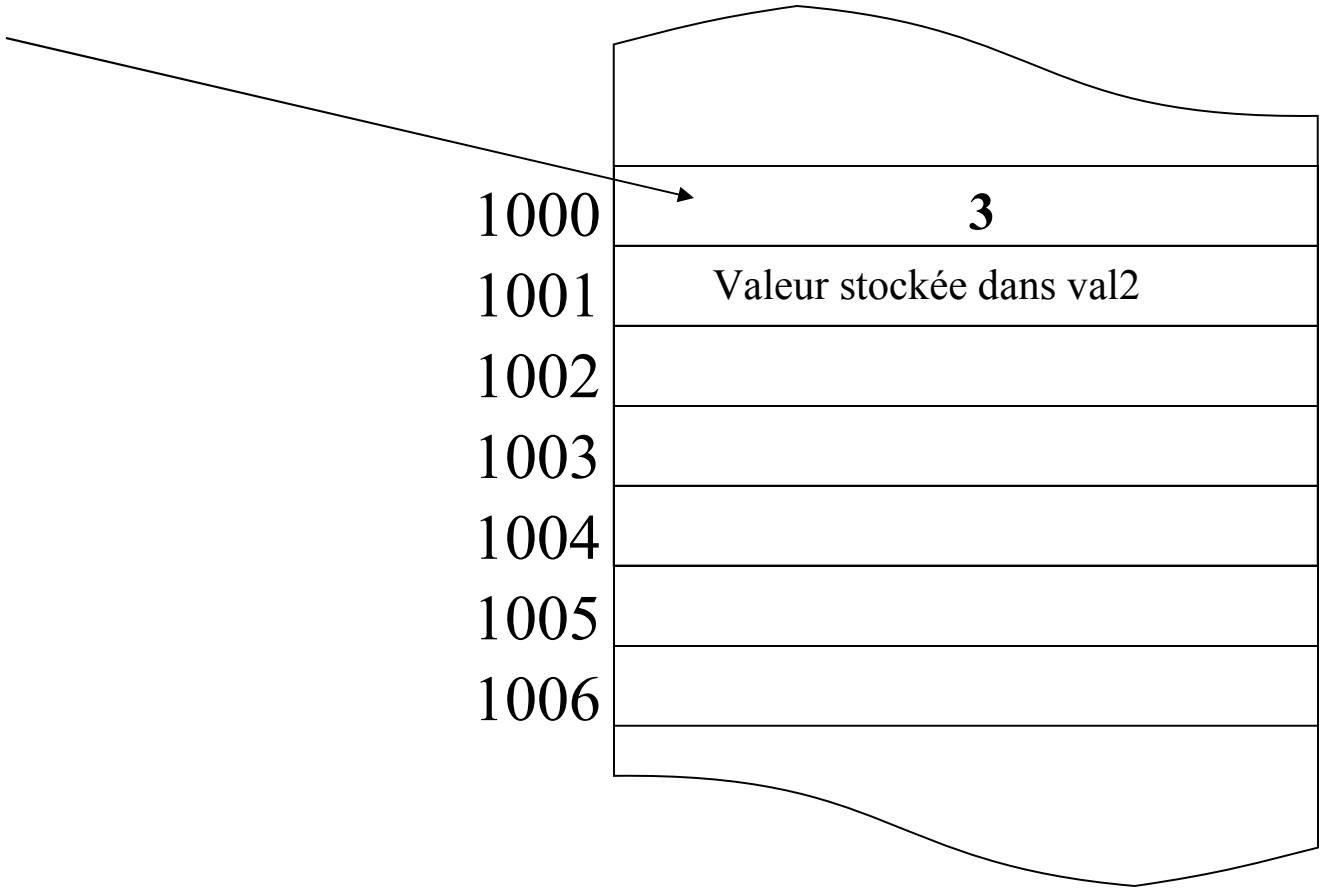
```
void main()
{
        int val1,val2;

        val1 = 3;
        val2 = -2*val1+4;

        if (val2 < val1)
                {
                printf("%d < %d\n",val2,val1);
                }
}
```

Le compilateur utilise sa table de correspondance : val1 ⇔ adresse 1000

Le nom de la variable "masque" son adresse

Les adresses

Effet de l'instruction `val1 = 3;` en mémoire



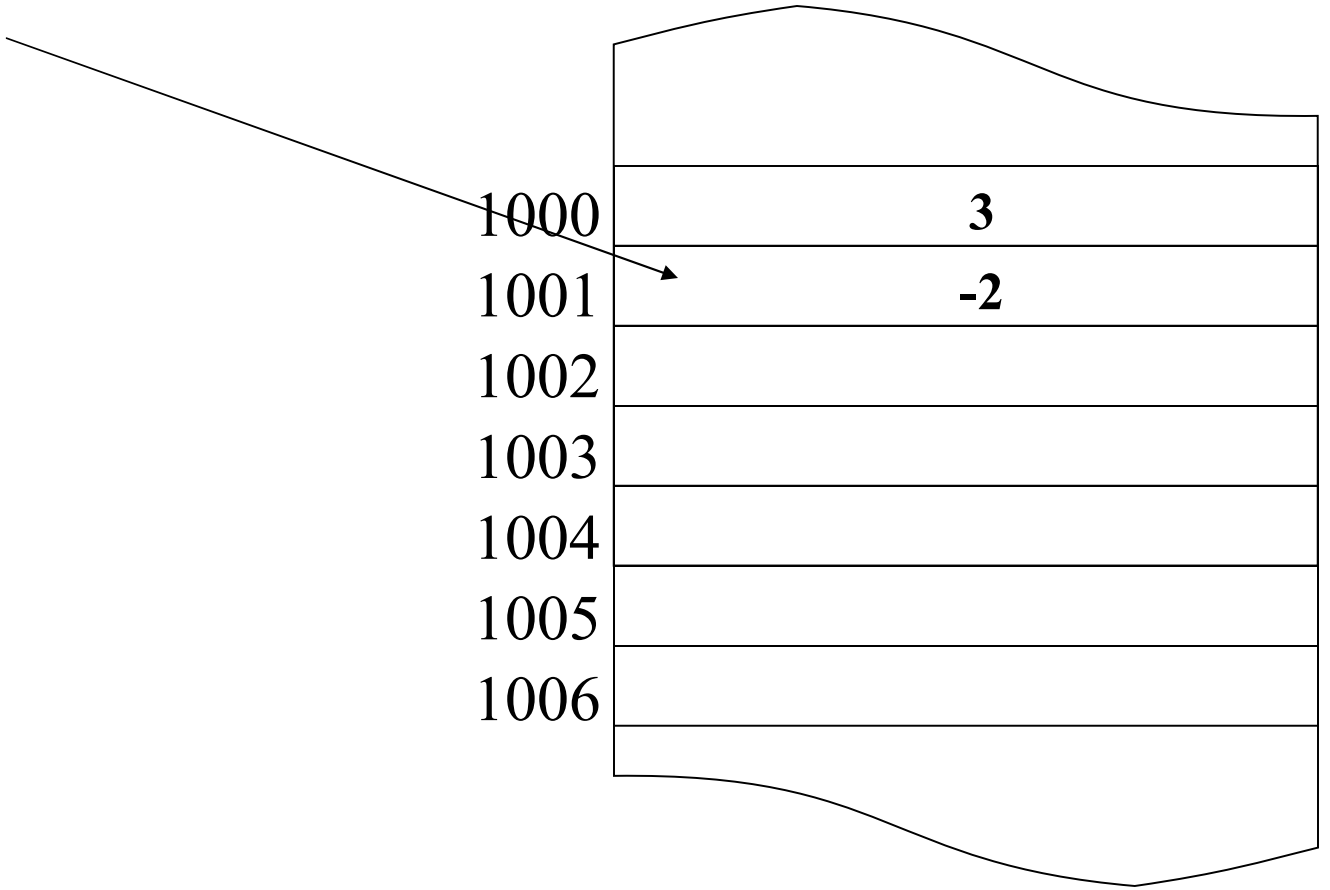
Les adresses

```
void main()
{
        int val1,val2;

        val1 = 3;
        val2 = -2*val1+4;

        if (val2 < val1)
                {
                printf("%d < %d\n",val2,val1);
                }
}
```

Le compilateur utilise sa table de correspondance pour val1 et val2

Les adresses

Effet de l'instruction `val2 = -2*val1+4;` en mémoire



Les adresses

On peut parfois avoir besoin de manipuler l'adresse d'une variable :

on peut y accéder en utilisant l'opérateur **&** (prise d'adresse) devant le nom de la variable.

`&nom_de_variable` se lit : adresse de la variable nom_de_variable.

Sur l'exemple précédent, après la déclaration de variable :

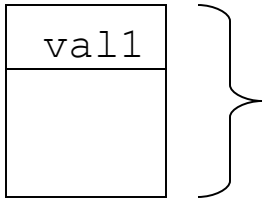
`val1` n'est pas initialisée (on ne connaît pas la valeur stockée à l'adresse correspondante, donnée par le compilateur)

`&val1` vaut 1000 : c'est l'adresse de val1

Les adresses

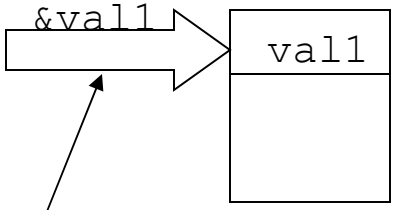
Une variable sera représentée par une boîte au dessus de laquelle est notée son nom, et qui contient sa valeur.

Exemple :



Représente la variable val1

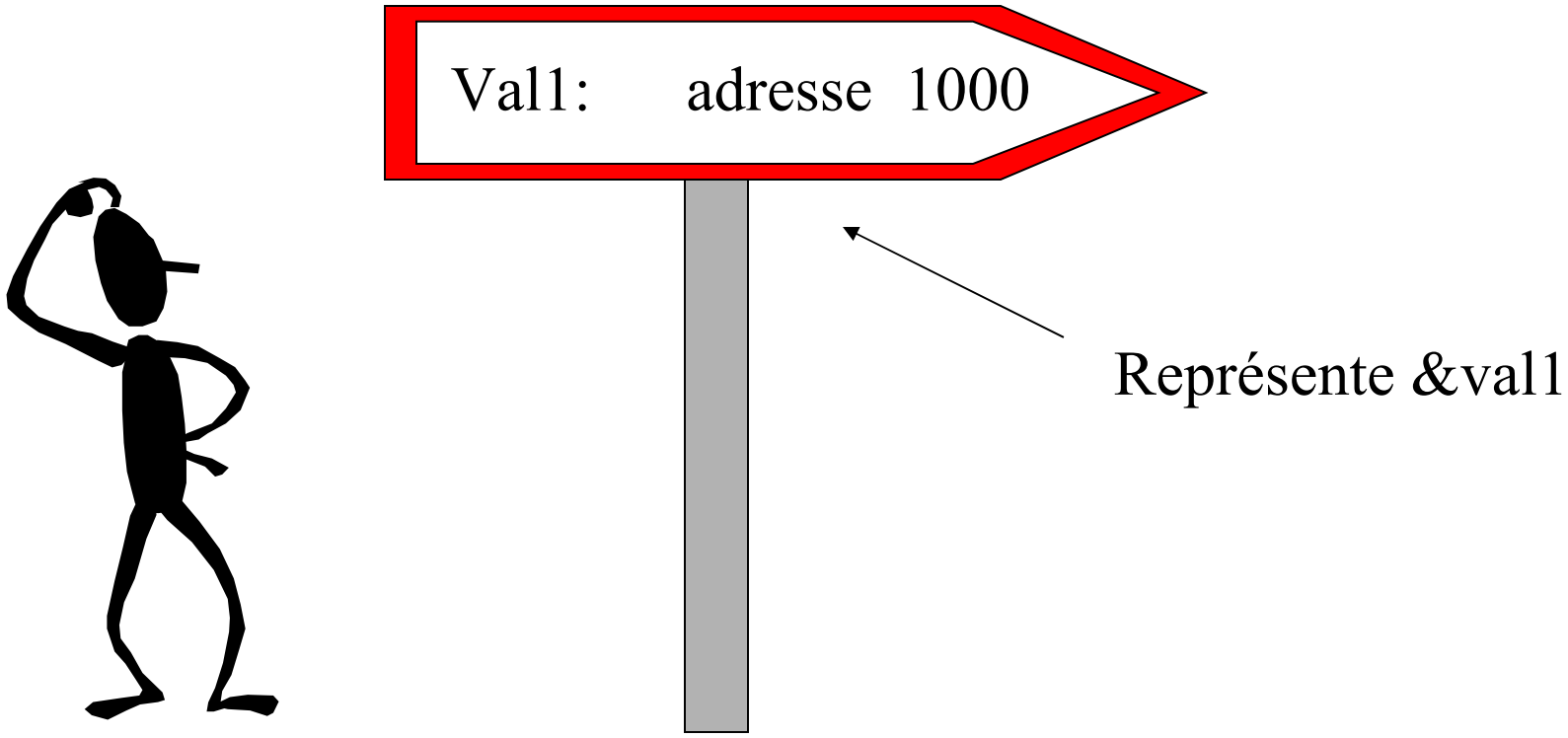
On symbolisera systématiquement une adresse par une flèche, pour indiquer que c'est le moyen utilisé pour trouver une variable dans la mémoire.



Représente
l'adresse de val1

Les adresses

On peut aussi voir cette flèche comme un panneau indicateur que l'on peut consulter.



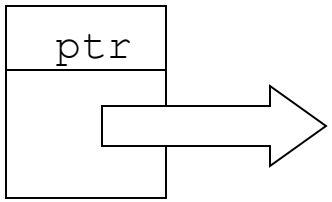
Les pointeurs

Les pointeurs sont des **variables** qui contiennent de tels panneaux ou flèches.

La valeur que contient une variable de type pointeur est une adresse (d'une variable par exemple).

Comme toute variable, on les symbolise par une boîte, mais cette boîte contient une flèche et non une valeur entière ou à virgule.

Soit **`ptr`** une variable de type pointeur (nous verrons la syntaxe de déclaration et d'utilisation plus tard). On la représente ainsi :



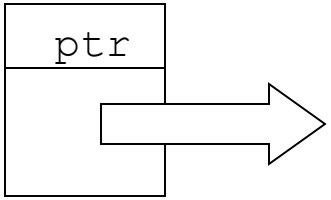
Les pointeurs

Les pointeurs sont des **variables** qui contiennent de tels panneaux ou flèches.

La valeur que contient une variable de type pointeur est une adresse (d'une variable par exemple).

Comme toute variable, on les symbolise par une boîte, mais cette boîte contient une flèche et non une valeur entière ou à virgule.

Soit ptr une variable de type pointeur (nous verrons la syntaxe de déclaration et d'utilisation plus tard). On le représente ainsi :



La prise de contenu

Raisonnement inverse :

on dispose de l'opérateur *****, qui placé **devant** un pointeur, permet d'accéder à la valeur contenue en mémoire à cette adresse

ne pas confondre avec la multiplication, qui a 2 opérandes

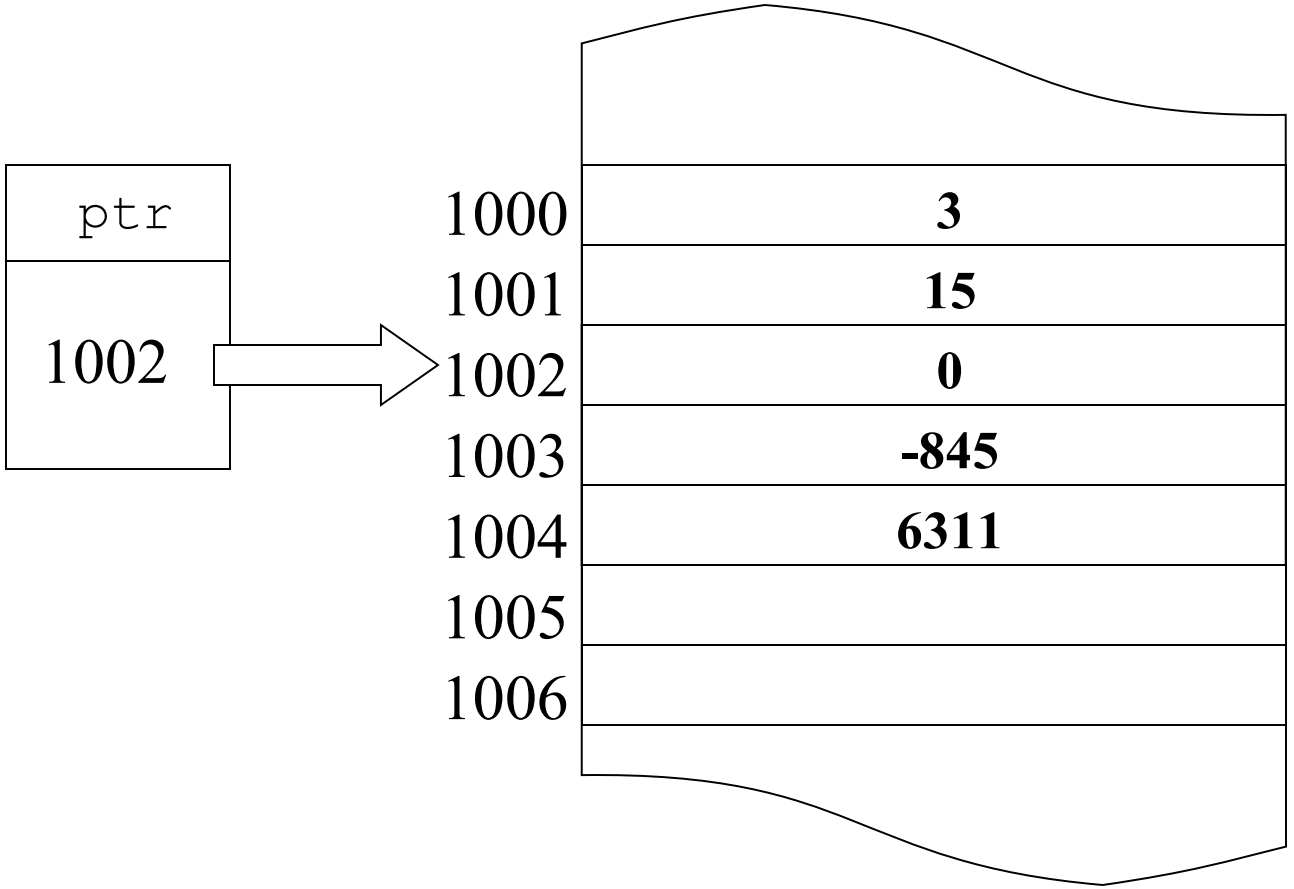
si **`ptr`** est un pointeur, alors **`*ptr`** est la valeur contenue en mémoire à l'adresse que contient **`ptr`**.

`*ptr` se lit : contenu de **`ptr`**.

Les pointeurs : illustration

Si ptr vaut 1002 :

*ptr est ce qui est contenu à l'adresse 1002 : 0



Pointeurs et types pointés

Lors de la prise de contenu : combien de cellules (ou d'octets) lire ?

Si c'est un **char** : 1 octet; **int** : 2 ou 4 octets; **float** : 4 octets

on doit connaître le type du contenu.

Un pointeur désigne un contenu typé :

pointeur sur **char**;

pointeur sur **int**;

pointeur sur **float**;

etc...

Type pointé précisé lors de la déclaration.

Déclaration

Syntaxe : prête à confusion

pointeur défini par le type du contenu qu'il pointe

exemple : déclarer une variable **`ptr`** comme un pointeur sur **int** ⇔ déclarer une variable **`ptr`** dont le contenu est de type **int**.

D'où la déclaration :

```
int *ptr;
```

se lit : le contenu de **`ptr`** (**`*ptr`**) est de type **int**

par abus de langage : **`ptr`** est un pointeur sur **int**;

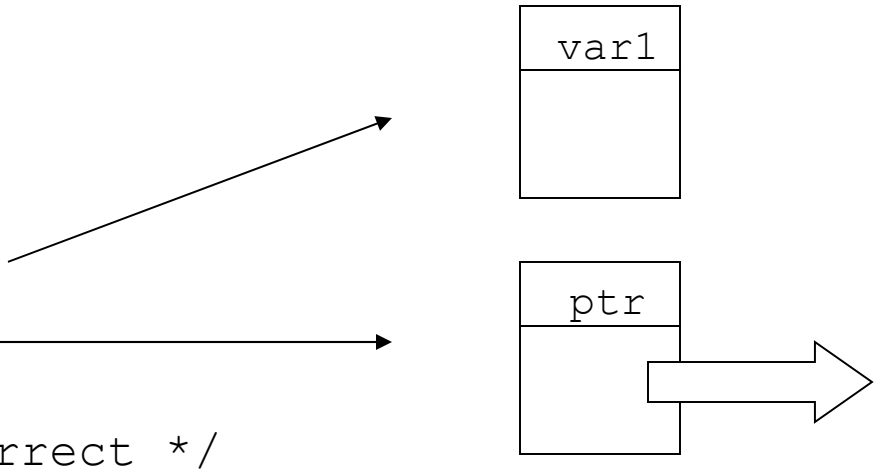
toujours lire `*` comme 'contenu' !

État du pointeur après déclaration

Programme exemple suivant : version 1

illustration avec boîtes et flèches

```
#include <stdio.h>

void main()
{
        short int var1;
        short int *ptr;

        var1 = 43; /* correct */
        *ptr = 5;  /* provoquera une erreur */
}
```

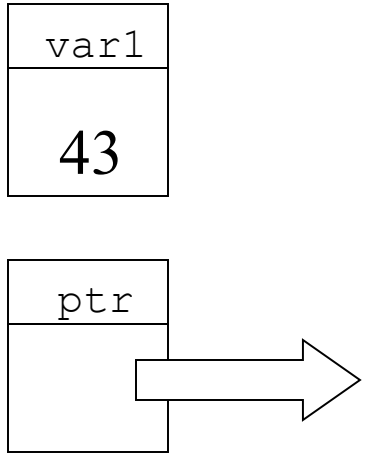


État du pointeur après déclaration

Programme exemple suivant : version 1

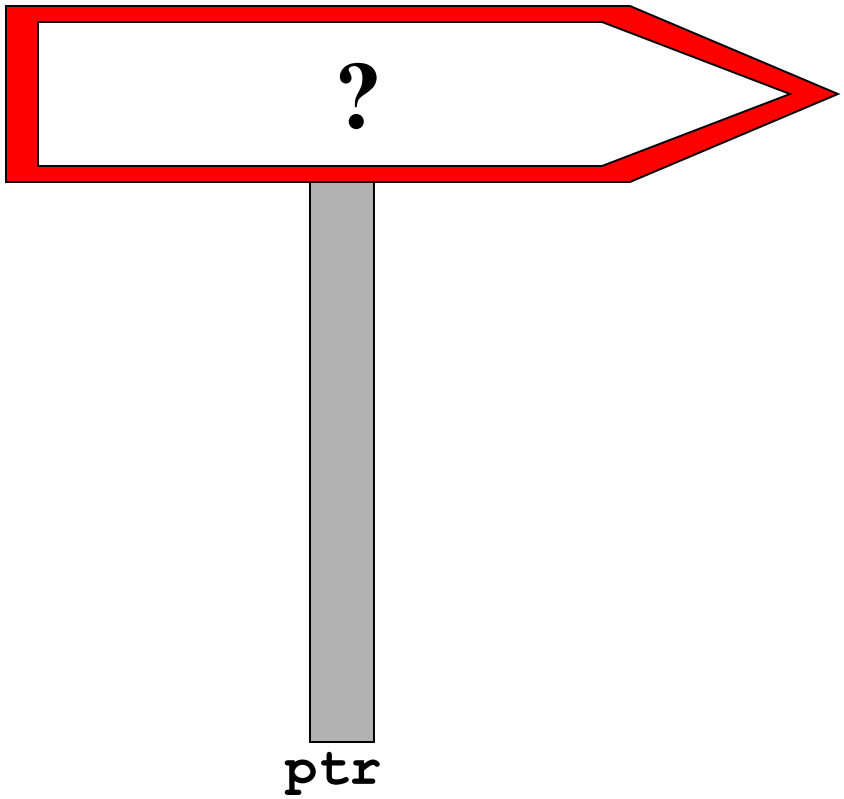
effet des instructions

```
#include <stdio.h>

void main()
{
        short int var1;
        short int *ptr;

        var1 = 43; /* correct */
        *ptr = 5;  /* provoquera une erreur */
}
```



État du pointeur après déclaration

`ptr` est juste le panneau (flèche), il ne pointe pas un endroit précis dans la mémoire



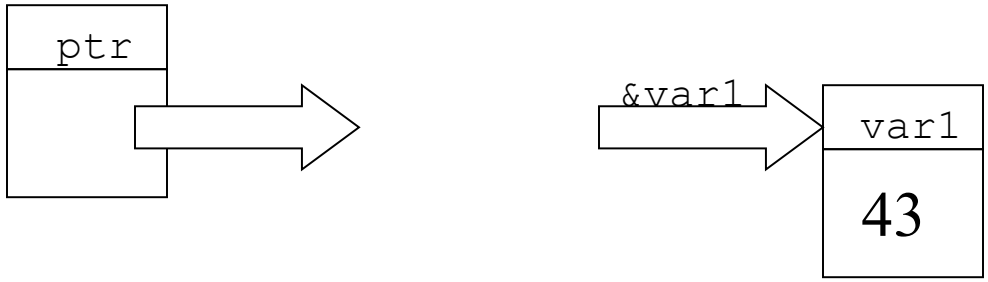
État du pointeur après déclaration

`*ptr` n'est pas défini : si l'on suit le panneau, on va n'importe où : erreur lors de l'exécution du programme.

Ajoutons la ligne suivante, entre `var1=43;` et `*ptr=5;`

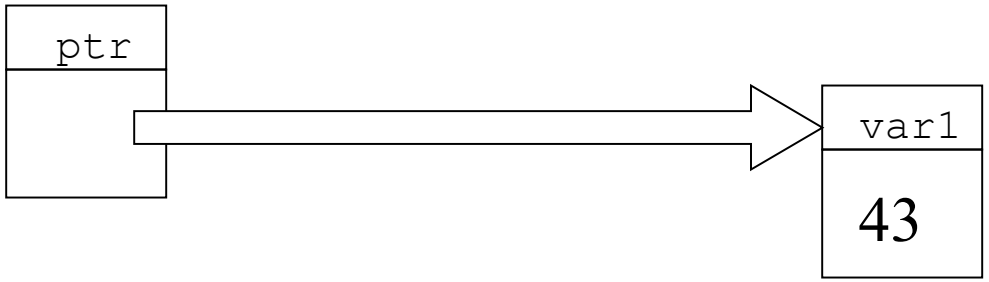
```
ptr = &var1;
```

détail de cette ligne avec boîtes et flèches :

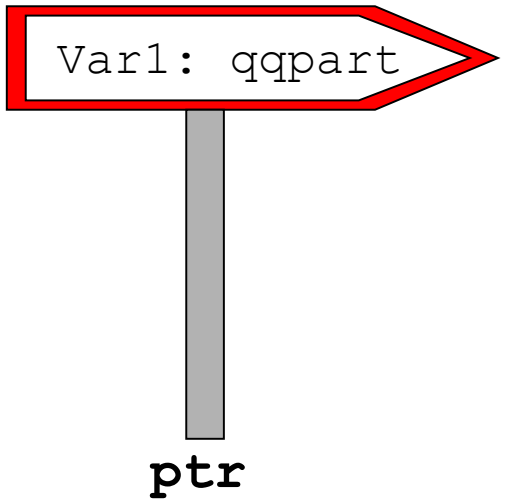


Égalité entres les flèches

État du pointeur après déclaration



Ou encore :

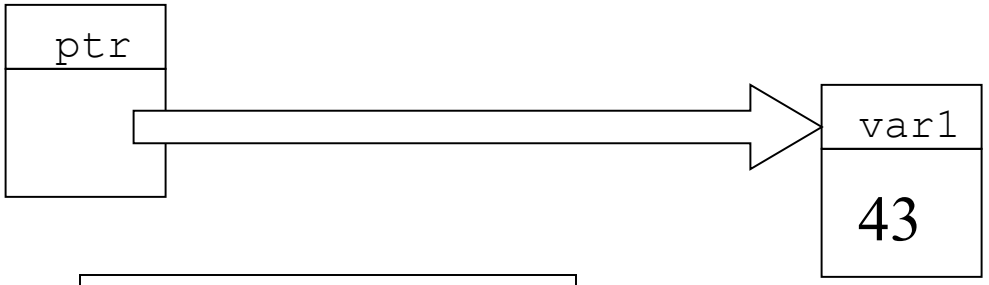


État du pointeur après déclaration

Maintenant, `ptr` pointe sur `var1`, le panneau indique une destination

que fait la ligne suivante : **`*ptr=5;`** ?

Avant cette ligne :



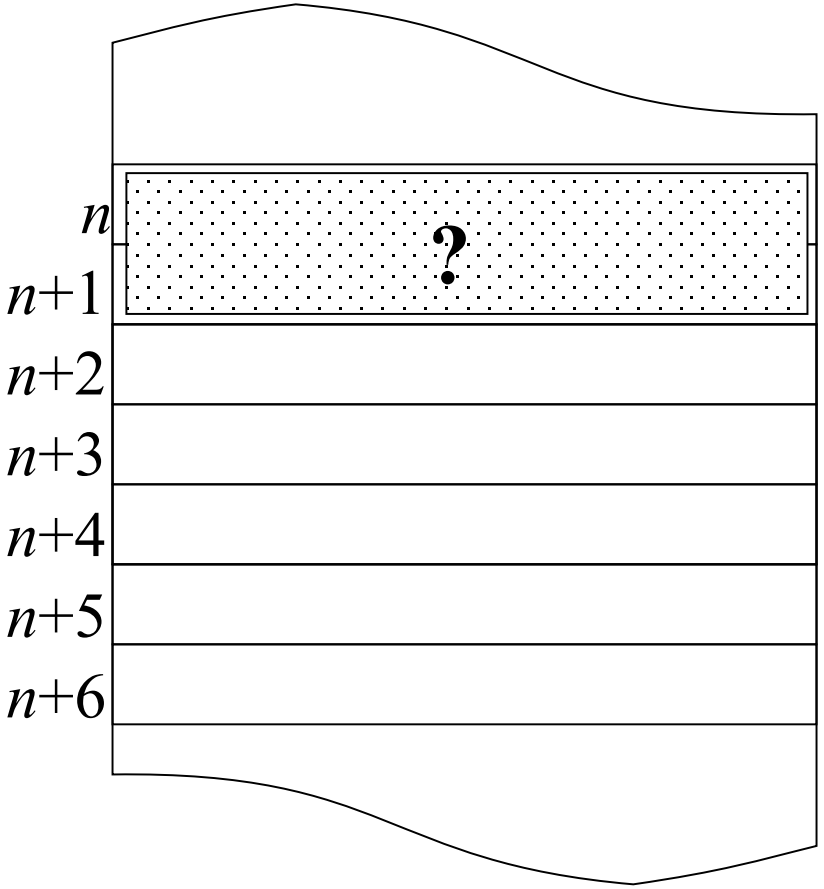
À vous de jouer

Après cette ligne :

État du pointeur après déclaration

Reprise de l'exemple avec le schéma de la mémoire :

```
short int var1;
short int *ptr;

var1 = 43;
ptr=&var1;
*ptr = 5;
```

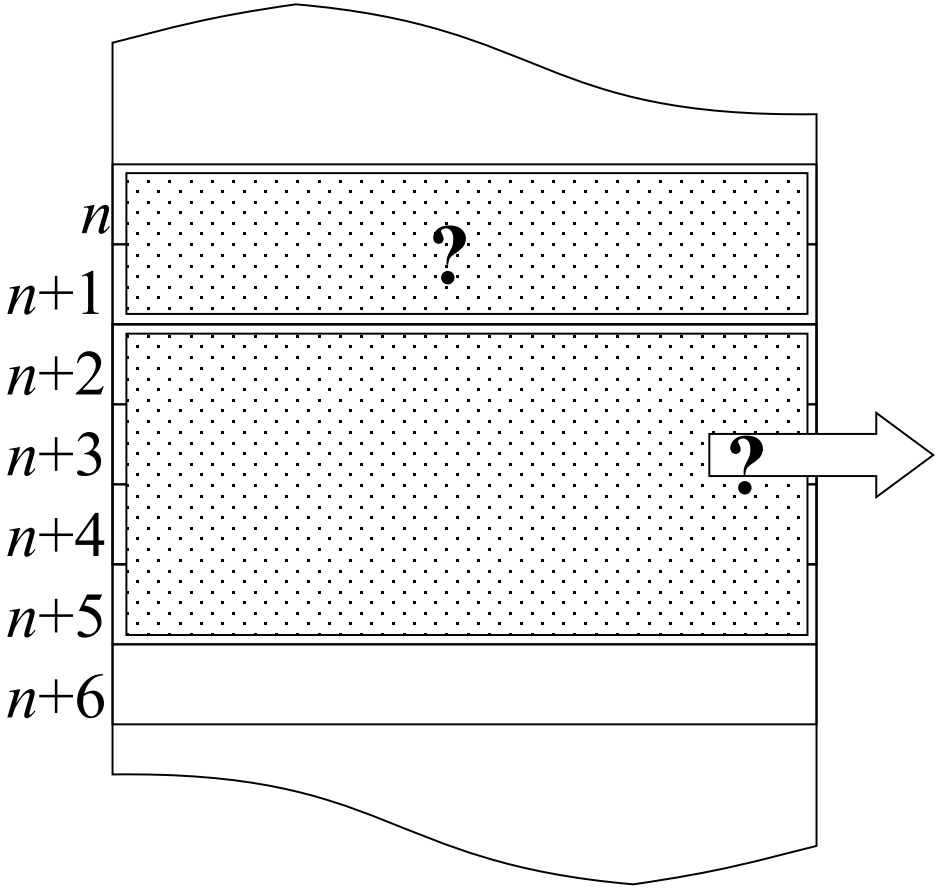


Effet : 2 octets utilisés pour stocker un **short int**.

État du pointeur après déclaration

Reprise de l'exemple avec le schéma de la mémoire :

```
short int var1;
short int *ptr;

var1 = 43;
ptr=&var1;
*ptr = 5;
```

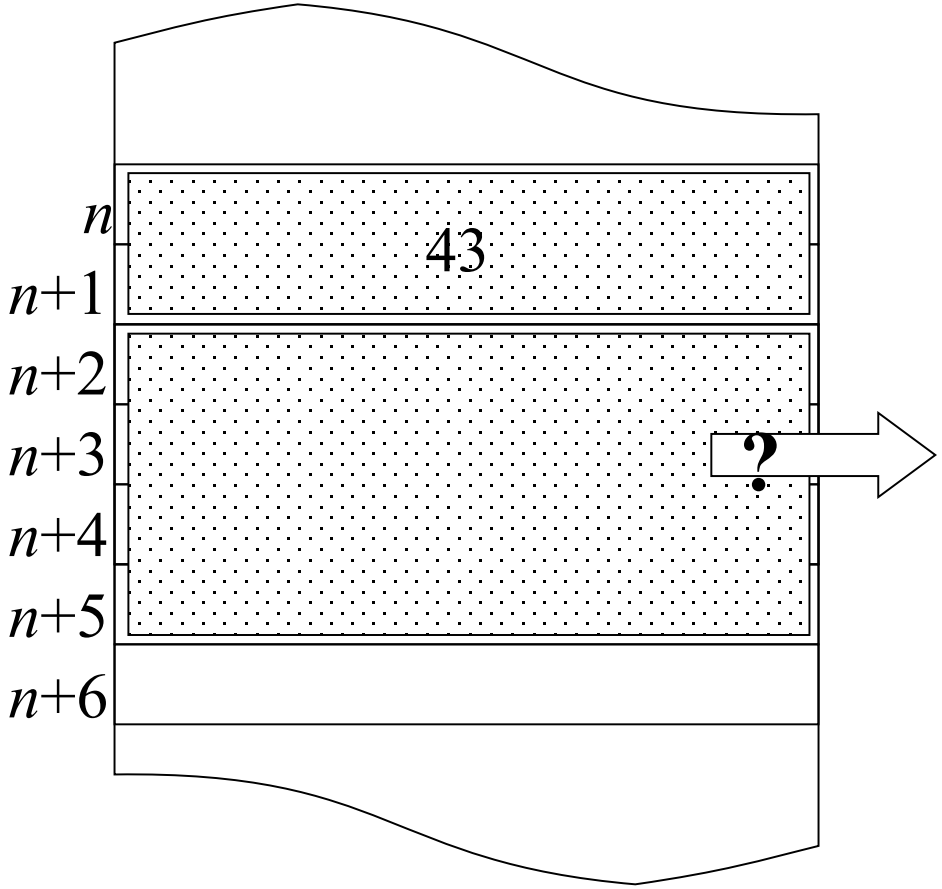


Effet : 4 octets utilisés pour stocker un pointeur vers un **short int**.

État du pointeur après déclaration

Reprise de l'exemple avec le schéma de la mémoire :

```
short int var1;
short int *ptr;

var1 = 43;
ptr=&var1;
*ptr = 5;
```

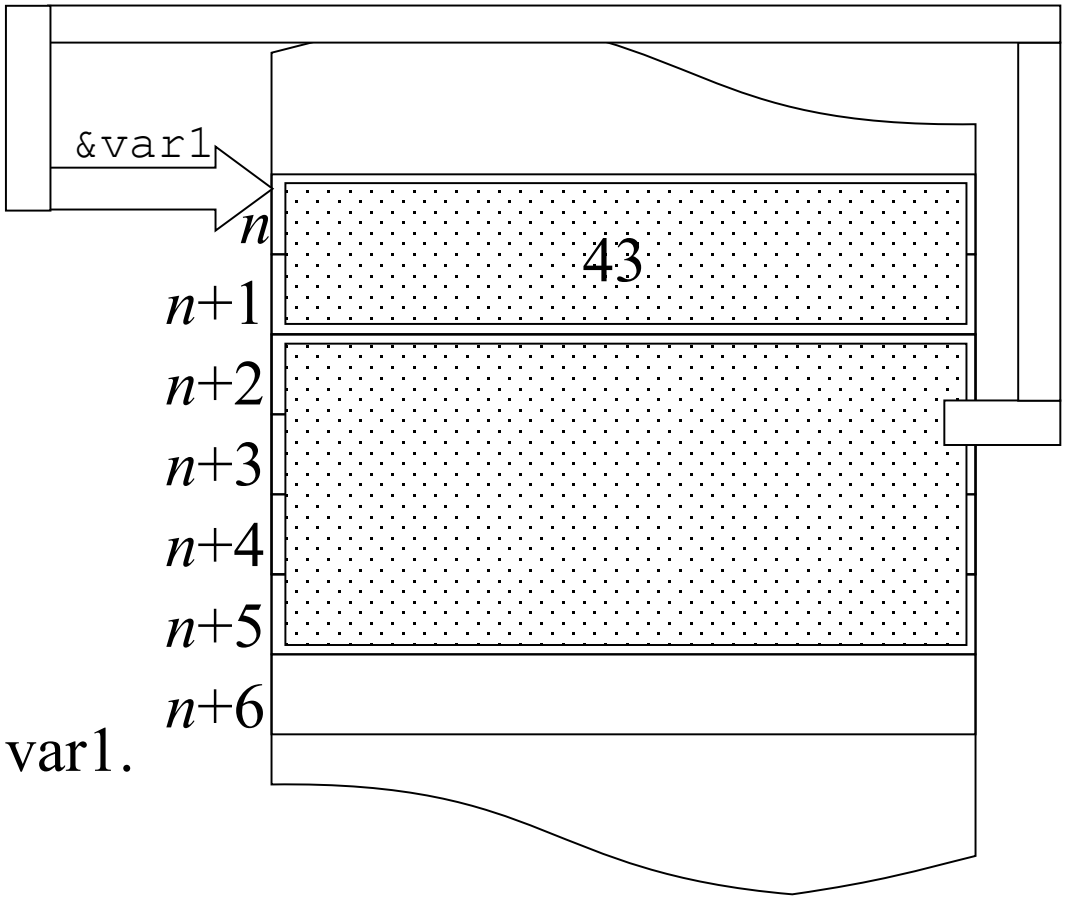


Effet : stocker 43 dans var1.

État du pointeur après déclaration

Reprise de l'exemple avec le schéma de la mémoire :

```
short int var1;
short int *ptr;

var1 = 43;
ptr=&var1;
*ptr = 5;
```

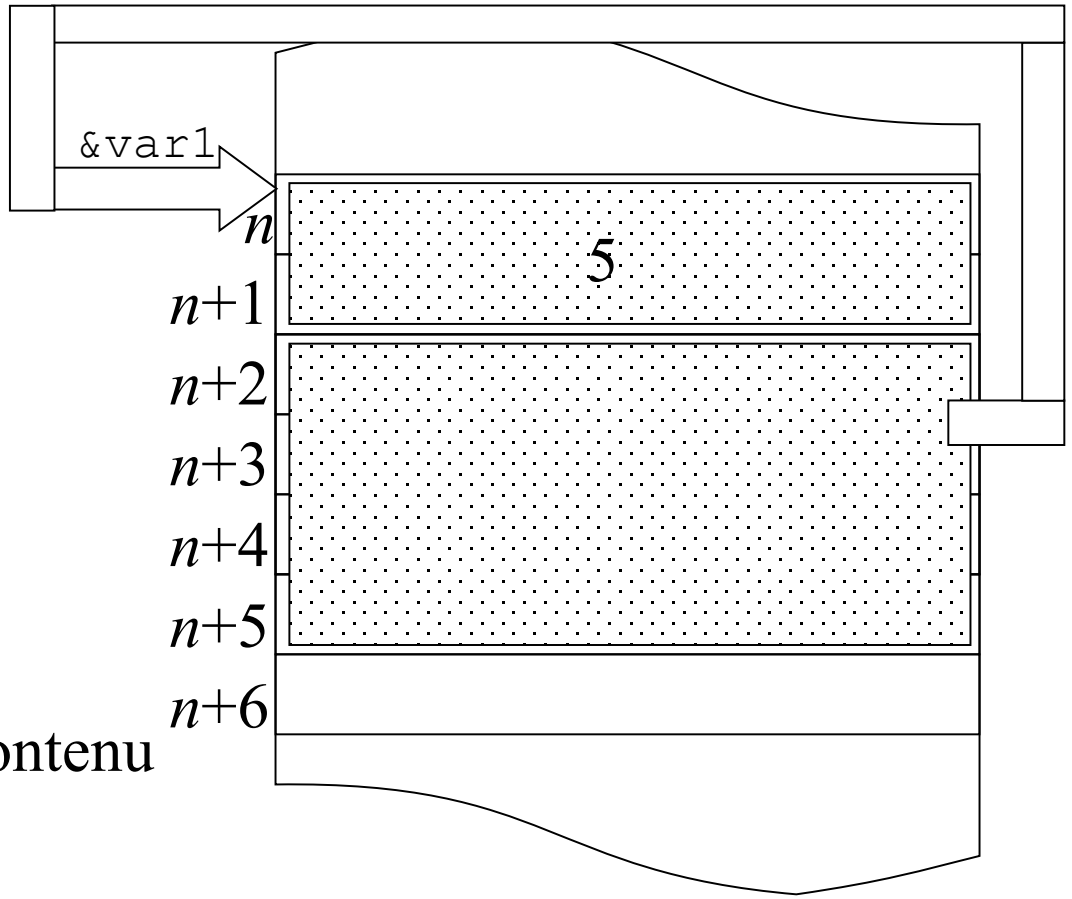


Effet : fait pointer ptr sur var1.

État du pointeur après déclaration

Reprise de l'exemple avec le schéma de la mémoire :

```
short int var1;
short int *ptr;

var1 = 43;
ptr=&var1;
*ptr = 5;
```



Effet : stocke 5 dans le contenu de ptr.

Manipulations

Que font les programmes suivants ? Donner une illustration.

```
#include <stdio.h>

void main()
{
	char a,b;
	char *p_ch;

	a=18;
	p_ch=&b;
	*p_ch=a;
	b=b+1;
	a=b;
	*p_ch=119;
}
```

Manipulations

Que font les programmes suivants ? Donner une illustration.

```
#include <stdio.h>

void main()
{
	float x_1,y_1;
	float *p_fl;

	x_1=3.14159;
	p_fl=&x_1;
	y_1 = 2.0*(*p_fl)*5.0;
	*p_fl= y_1 - x_1;
}
```

Manipulations

Soient les instructions suivantes :

```
int a;
int *ptr
a=4;
ptr=&a;
```

opérations avec contenus :

```
*ptr=*ptr+1;        /* effet connu */
```

opérations avec pointeurs :

```
ptr = ptr+1;        /* quel effets ??? */
```

Manipulations

opérations avec contenus :

```
*ptr=*ptr+1;
```

effet : calcule : contenu de `ptr` +1 : 4 +1 → 5, rangé dans contenu de `ptr`

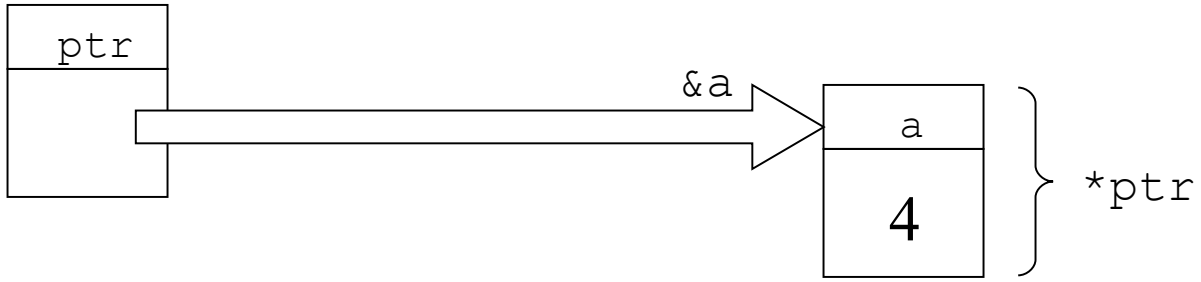
opérations avec pointeurs :

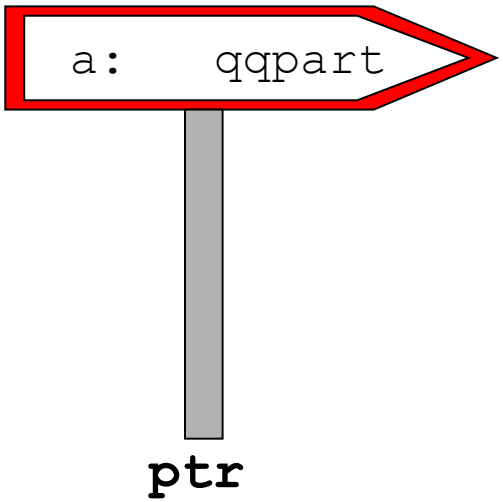
```
ptr = ptr+1;        /* quel effet ??? */
```

`ptr` désigne la flèche.

Manipulations

État des variables :

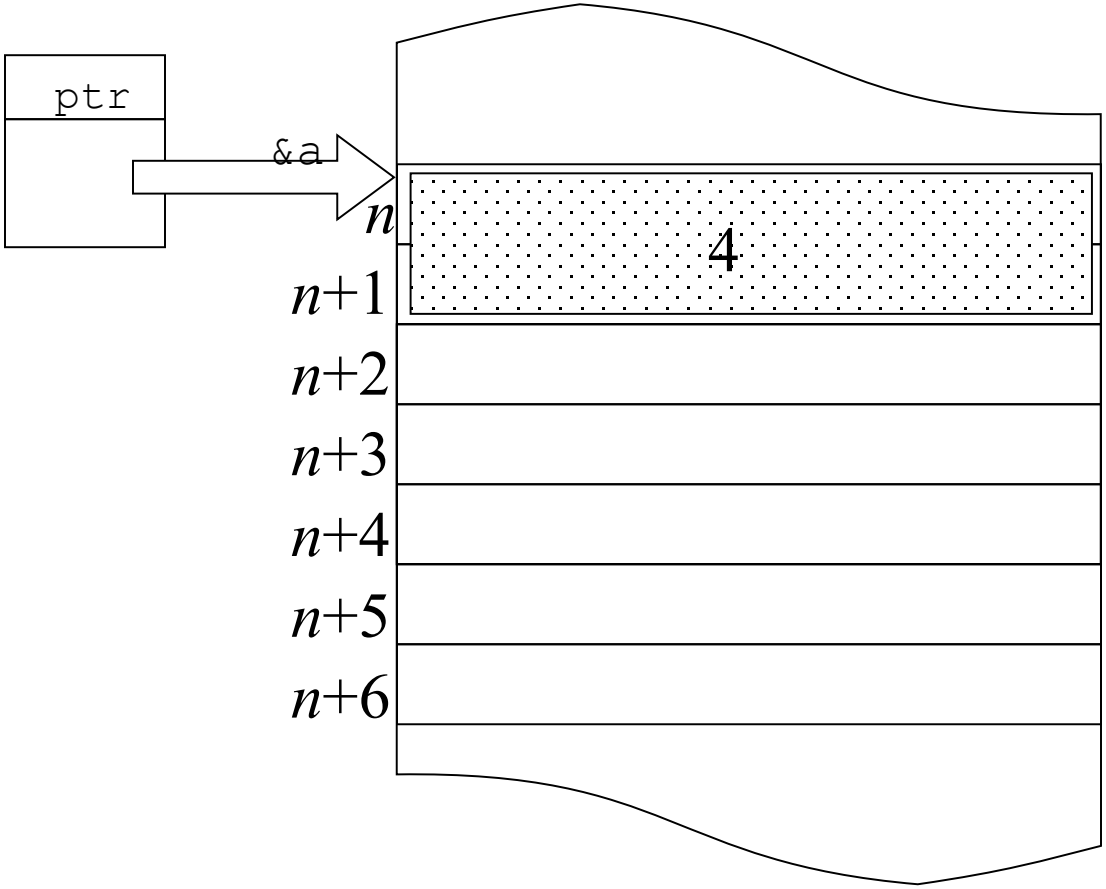




a est situé quelque part en mémoire

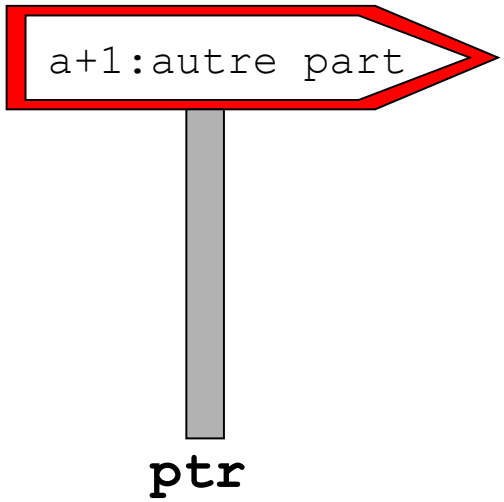
Manipulations

a est situé quelque part en mémoire



Manipulations

Faire **`ptr = ptr+1;`** change la flèche elle-même.



Manipulations

Arithmétique des pointeurs :

<u>ajouter un entier à un pointeur :</u> ***p*** pointeur et ***n*** entier.

p=p+n;

effet : *p* (et non son contenu) pointe *n.t* octets plus loin dans la mémoire (*t* : taille du type en octets).

<u>soustraction de 2 pointeurs :</u> ***p*** et ***q*** pointeurs sur un même type;

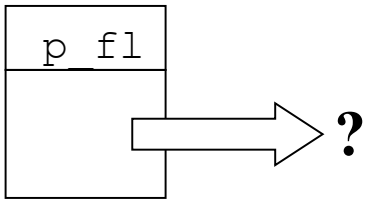
p-q;

effet : donne la taille de la mémoire située entre ***p*** et ***q***.

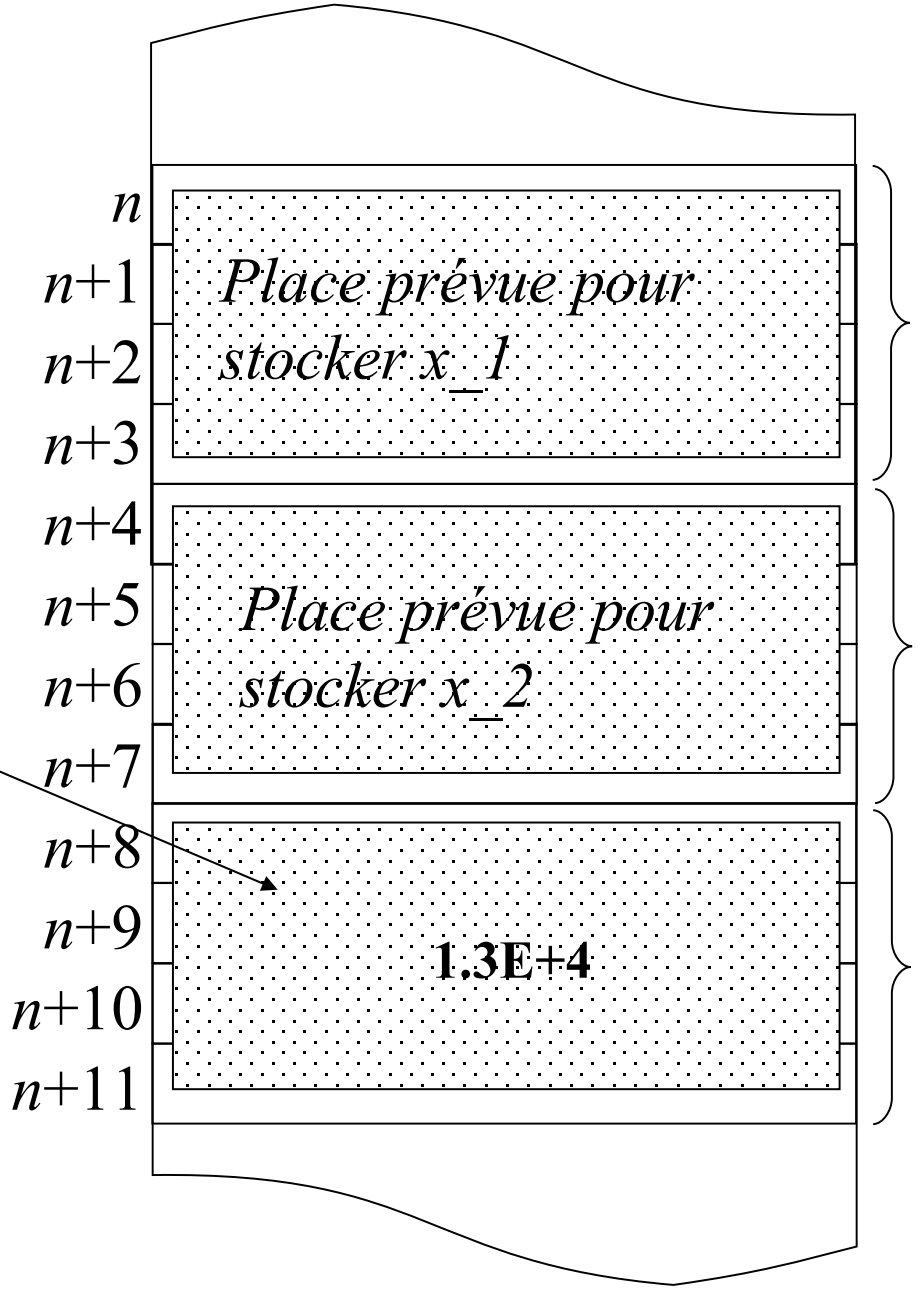
Manipulations

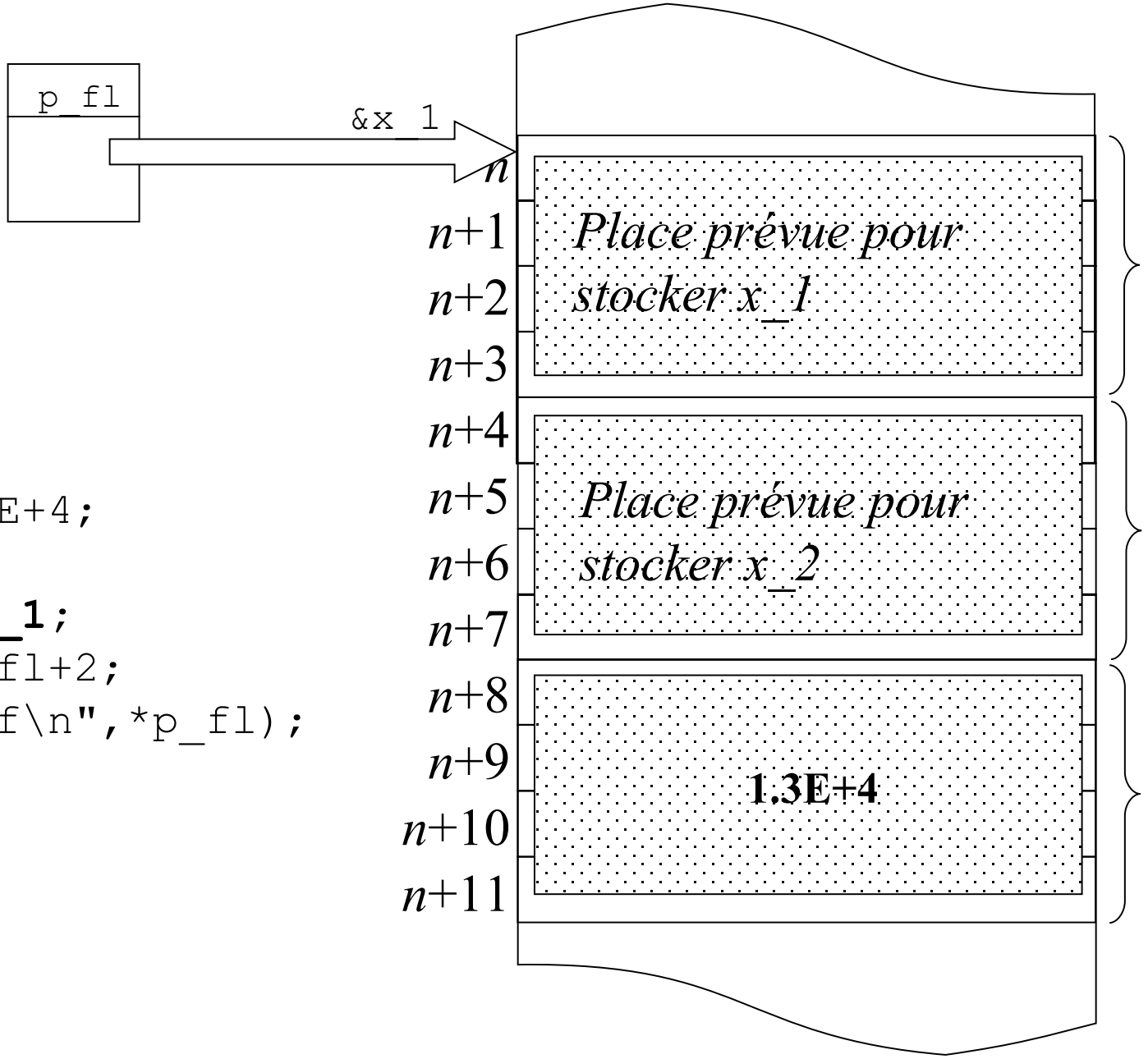
Programme exemple : *rappel : 1 variable float occupe 4 octets*

```
#include <stdio.h>

void main()
{
      float x_1, x_2, x_3;
      float *p_fl;

      x_3 = 1.3E+4;

      p_fl=&x_1;
      p_fl = p_fl+2;
      printf("%f\n",*p_fl);
}
```



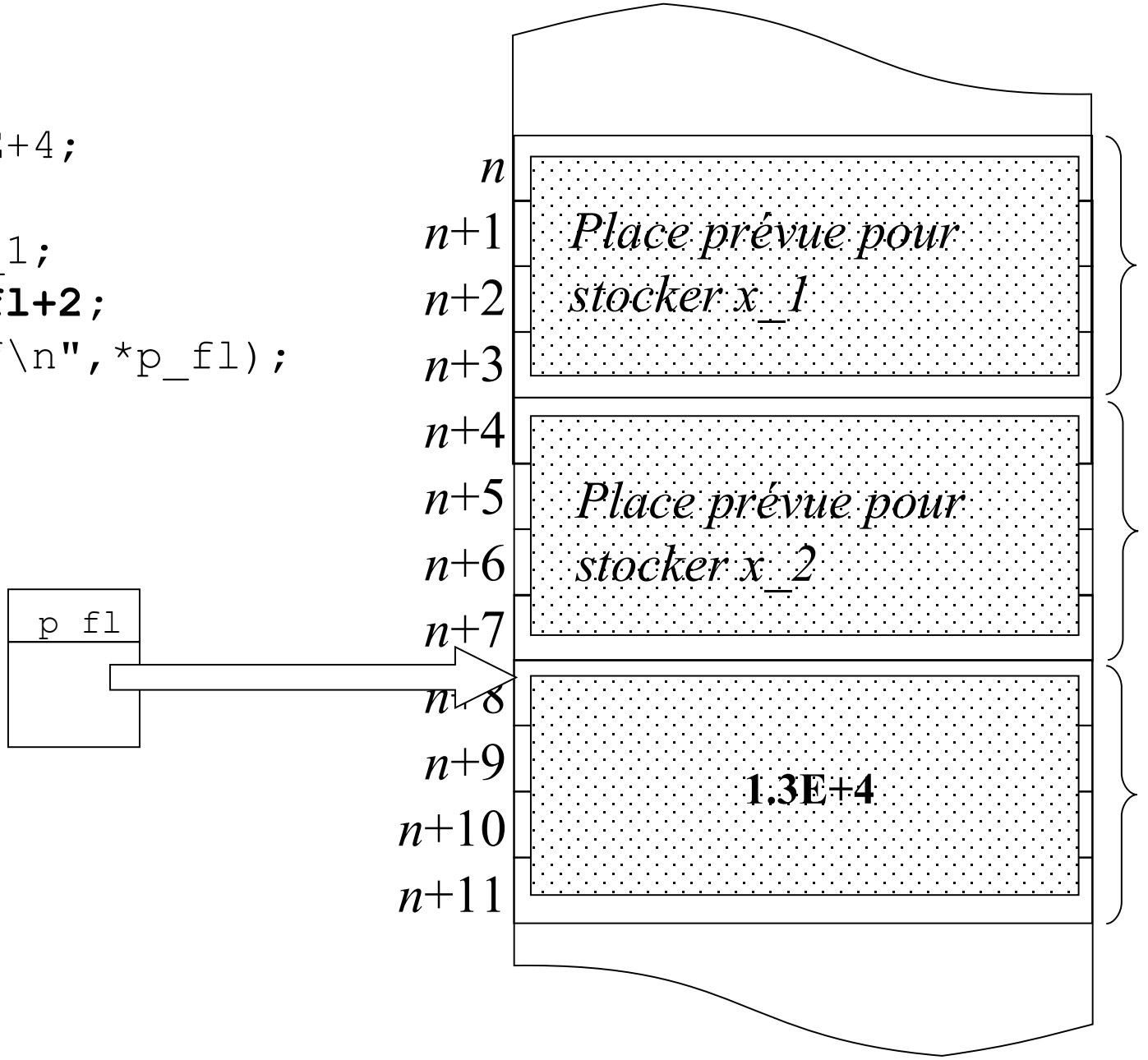

```
x_3 = 1.3E+4;

p_fl = &x_1;
p_fl = p_fl+2;
printf("%f\n",*p_fl);
```






```
x_3 = 1.3E+4;

p_fl = &x_1;
p_fl = p_fl+2;
printf("%f\n",*p_fl);
```

```
x_3 = 1.3E+4;

p_fl = &x_1;
p_fl = p_fl+2;
printf("%f\n",*p_fl);
```



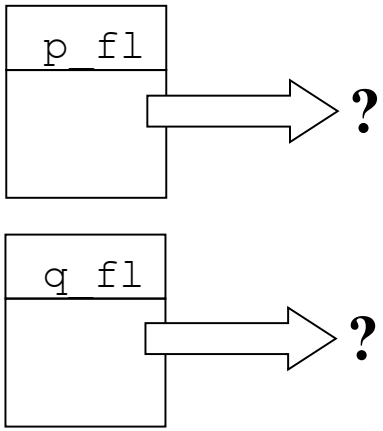
```
x_3 = 1.3E+4;

p_fl = &x_1;
p_fl = p_fl+2;
printf("%f\n",*p_fl);
```

1.3E+4

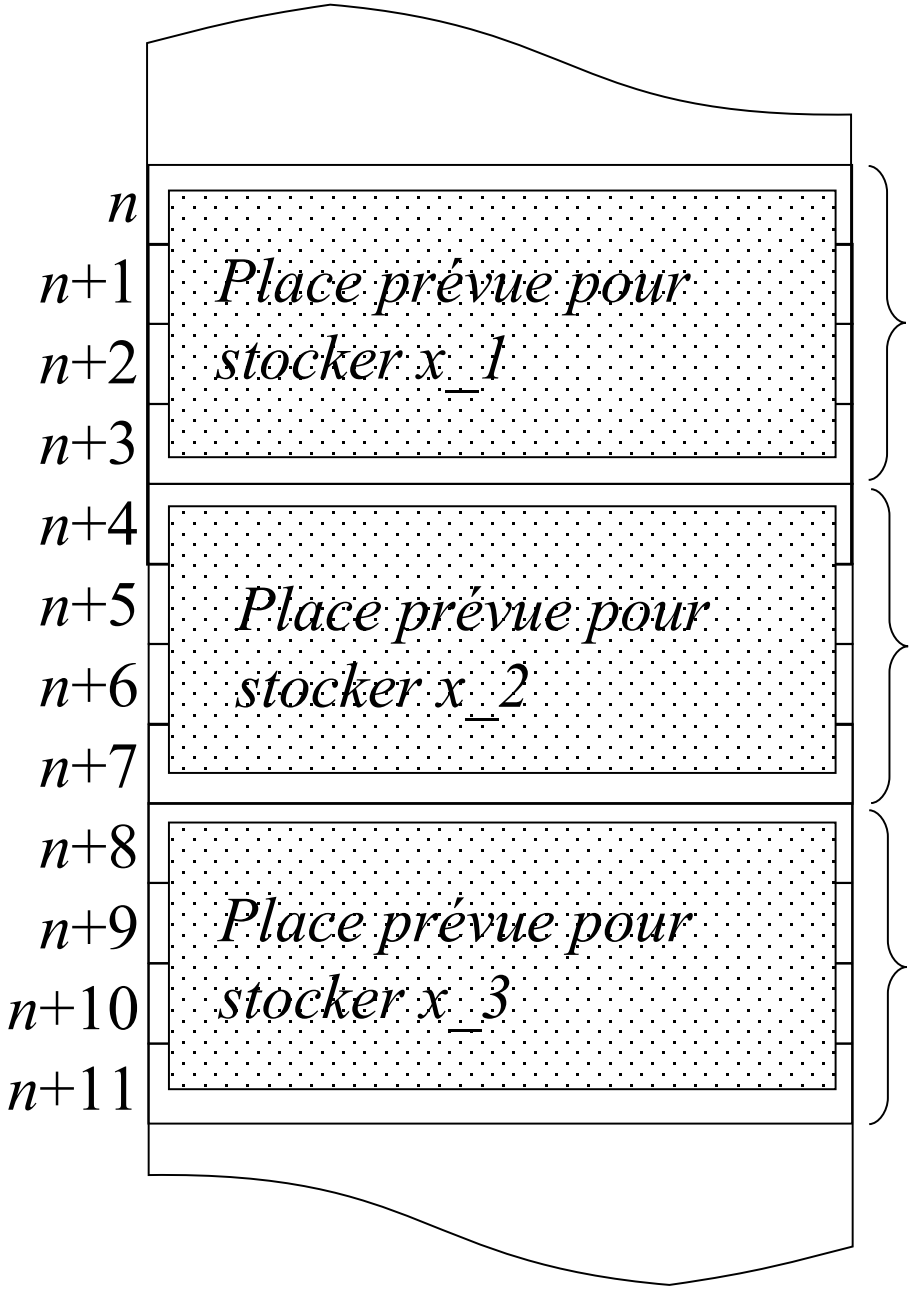
Manipulations

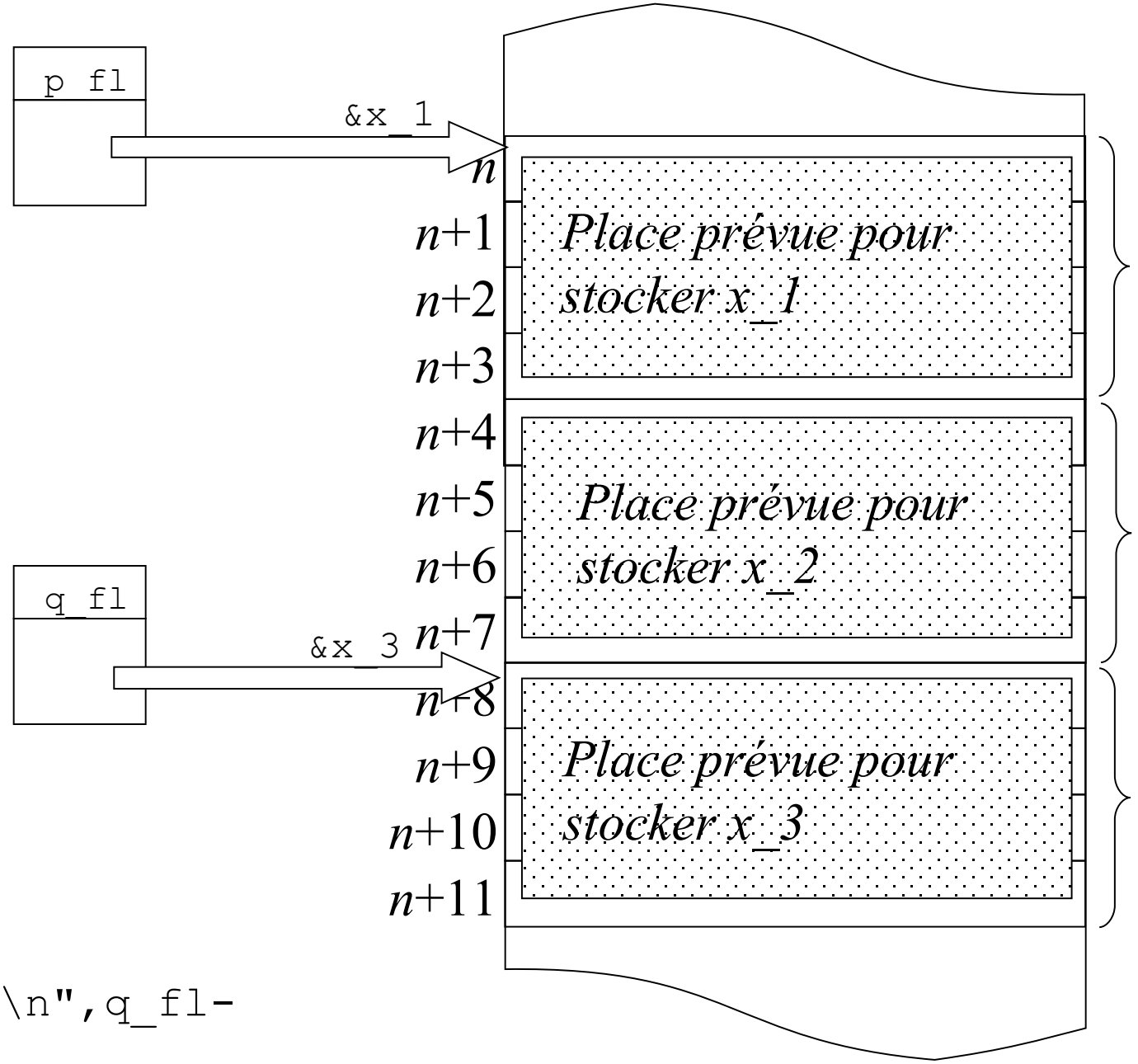
Programme exemple : *rappel : 1 variable float occupe 4 octets*

```
#include <stdio.h>

void main()
{
	float x_1, x_2, x_3;
	float *p_fl;
	float *q_fl;

	p_fl=&x_1;
	q_fl=&x_3;
	printf("%d\n",q_fl-p_fl);
}
```




```
p_fl=&x_1;
q_fl=&x_3;

printf("%d\n",q_fl-
p_fl);
```






```
p_fl=&x_1;
q_fl=&x_3;

printf("%d\n",q_fl-
p_fl);
```

Manipulations

```
p_fl=&x_1;
q_fl=&x_3;
printf("%d\n",q_fl-p_fl);
```

```
2
```

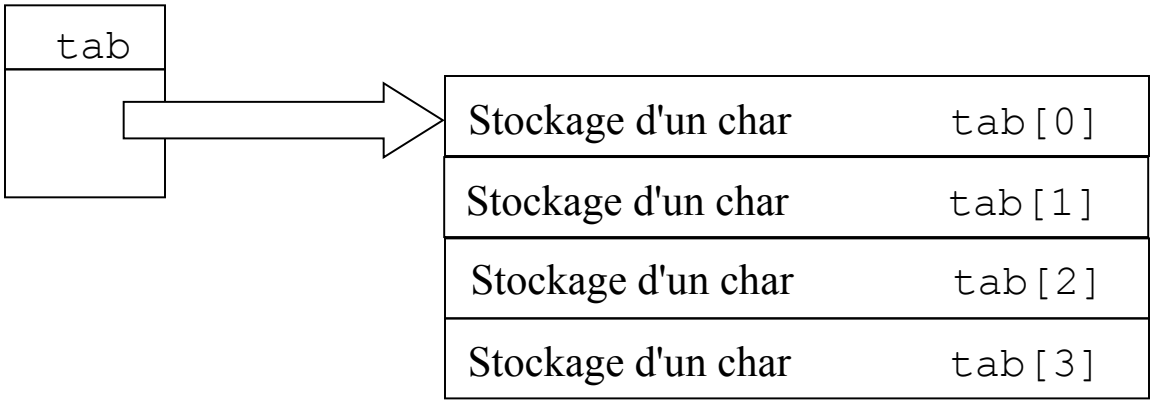
Pointeurs et tableaux

Relation entre tableaux et pointeurs

un tableau est un pointeur !

Donne accès au premier élément du tableau…⇔ pointe sur le premier élément du tableau !

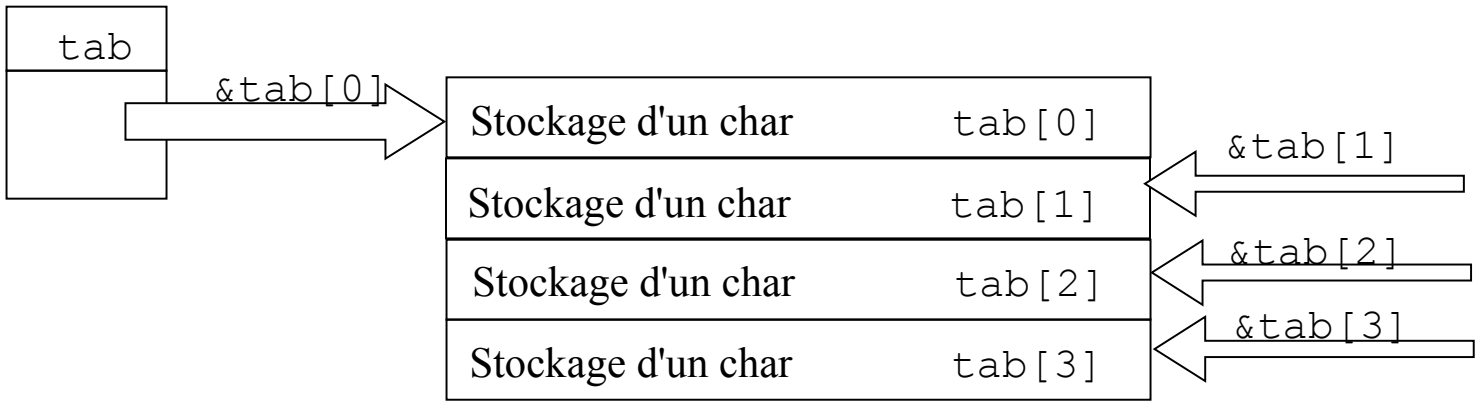
char tab[4]; se représente ainsi :



Pointeurs et tableaux

En fait `tab` = `&tab[0]`

par contre `tab` est constant : non modifiable



On a : **`tab+i` = `&tab[i]`**

Allocation dynamique

Un tableau est alloué de manière statique : nombre d'éléments constant.

Alloué lors de la compilation (avant exécution)

problème pour déterminer la taille optimale, donnée à l'exécution

surestimation et perte de place

de plus, le tableau est un pointeur constant.

Il faudrait un système permettant d'allouer un nombre d'éléments connu seulement à l'exécution : c'est l'**allocation dynamique.**

Allocation dynamique

Faire le lien entre le pointeur non initialisé (le panneau vide) et une zone de mémoire de la taille que l'on veut.

On peut obtenir cette zone de mémoire par l'emploi de **malloc**, qui est une fonction prévue à cet effet.

Il suffit de donner à **malloc** le nombre d'octets désirés (attention, utilisation probable de sizeof), et **malloc renvoie un pointeur de type *void** sur la zone de mémoire allouée**.

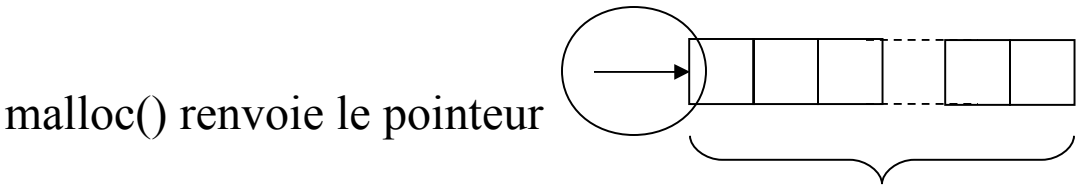
Si malloc n'a pas pu trouver une telle zone mémoire, il renvoie NULL.

Appel par : malloc(nombre_d_octets_voulus);

Allocation dynamique

Symbolisation de l'effet de malloc:

si on utilise par exemple malloc(n), on a :



Pour accéder à cette zone, il faut impérativement l'affecter à un pointeur existant. On trouvera donc **toujours** malloc à droite d'un opérateur d'affectation.

Il ne faut pas oublier de transtyper le résultat de malloc() qui est de type void* en le type du pointeur auquel on affecte le résultat.

Allocation dynamique

Exemples d'utilisation :

allocation dynamique pour un pointeur vers des entiers, (analogue à un tableau d'entiers). On demandera à l'utilisateur le nombre d'éléments qu'il souhaite, puis on fait l'allocation dynamique correspondante :

tableau de la taille requise, pas de perte de mémoire !

```
#include <stdio.h>
void main()
{
	int *pt_int;
	int nbElem;

	printf("combien d'elements dans le tableau ?:");
	scanf("%d",&nbElem);
}
```

Allocation dynamique

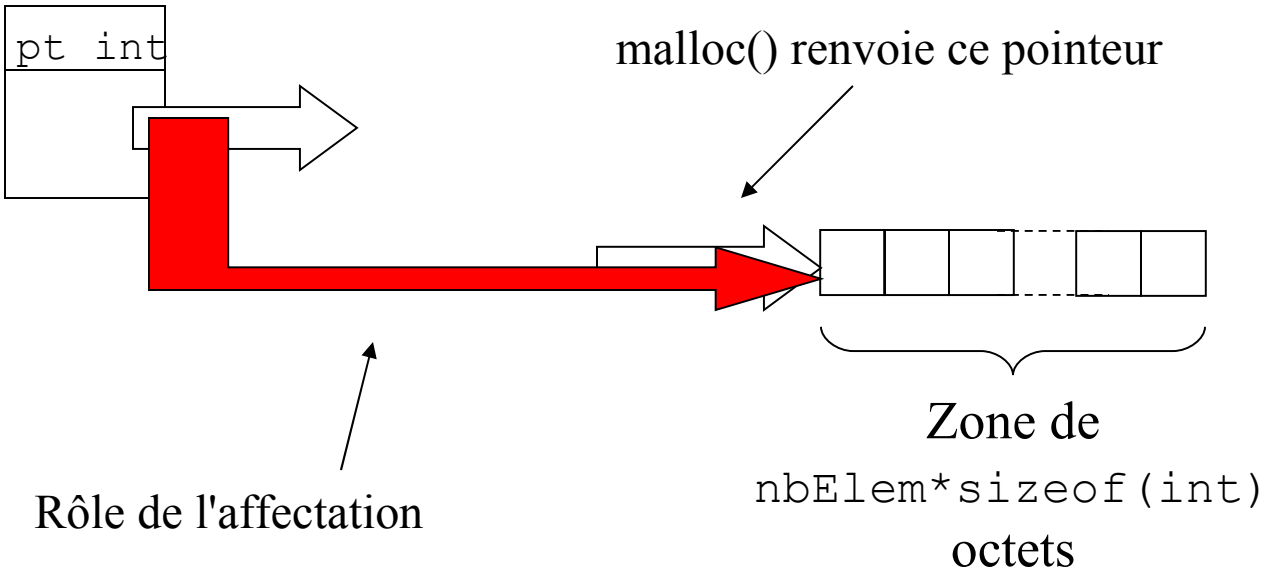
```
pt_int = (int *)malloc(nbElem*sizeof(int));
```

détail de cette ligne : analyse de l'expression à droite de l'opérateur d'affectation :
(int *) : transtypage : car malloc() donne un pointeur void *, et pt_int est de type int *
malloc : appel à la fonction
nbElem*sizeof(int) : n'oublions pas que malloc reçoit un nombre d'octets a allouer ! Ici, on veut allouer nbElem élements, qui sont chacun de type int ! Or un int occupe plus d'un octet. Il occupe sizeof(int) octets !
Donc le nombre total d'octets à demander est : **nombre d'éléments * taille de chaque élément en octets**.

Allocation dynamique

```
pt_int = (int *)malloc(nbElem*sizeof(int));
```

avec la symbolisation déjà vue pour les pointeurs :



Allocation dynamique

Vérifier le fonctionnement : si malloc() à donné NULL, l'allocation a échoué. Toujours tester la valeur du pointeur affecté après l'emploi de malloc().

```
if (pt_int == NULL)
{
	printf("allocation n'a pas fonctionne\");
}
else
{
	/* suite du programme */
}
```

on peut maintenant utiliser pt_int comme un tableau d'entiers, avec la notation [] !

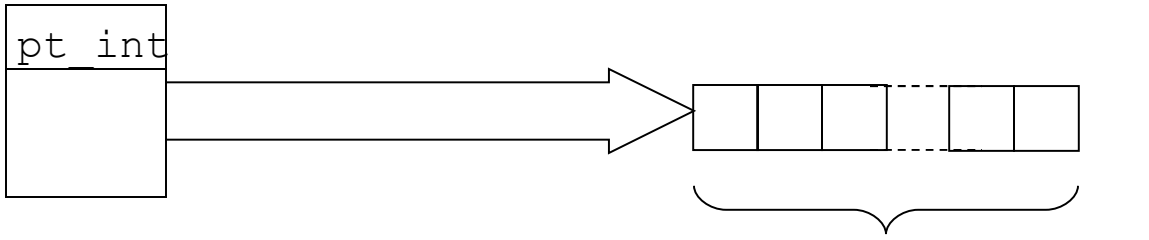
Libération de l'espace alloué

Lorsque la mémoire allouée dynamiquement n'est plus utile (le plus souvent, à la fin d'un programme, il est nécessaire de la libérer : la rendre disponible pour le système d'exploitation.

Fonction free qui réalise le contraire de malloc().

free(pointeur_vers_la_zone_allouée);

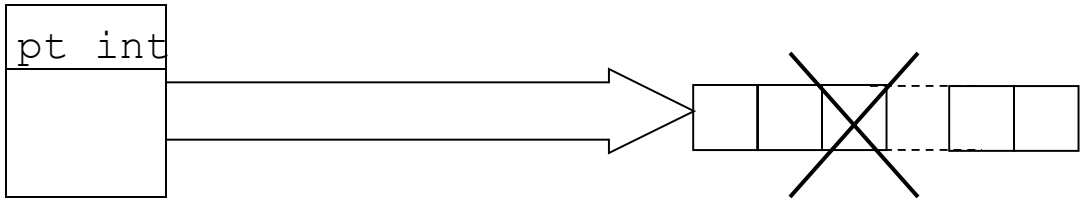
on ne peut libérer que des zones allouées dynamiquement : pas de free avec un tableau statique, même si le compilateur l'accepte.



Libération de l'espace alloué

Si on écrit : free(pt_int);

effet :



pt_int pointe toujours au même endroit de la mémoire, mais on ne peut plus l'utiliser.

À chaque malloc() doit correspondre un free() dans un programme !

Sinon, l'ordinateur le fait à votre place : ne lui faites pas confiance !