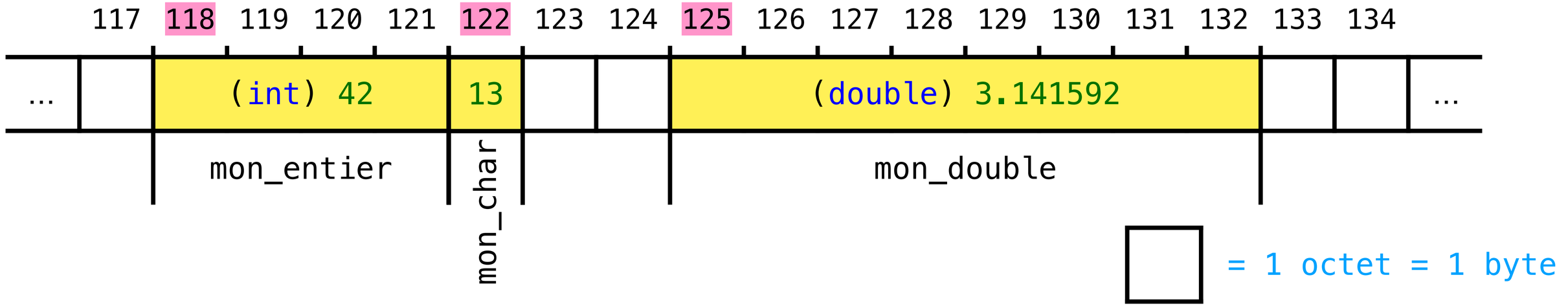
Pointeurs

et tableaux

ICC-C Cours 7

Rappels

- Chaque variable est stockée à un emplacement de mémoire fixe
- Selon son type, une variable occupe un certain nombre d'octets en mémoire
- Chaque octet de la mémoire d'un programme a une adresse
- L'adresse d'une variable est l'adresse de son premier octet

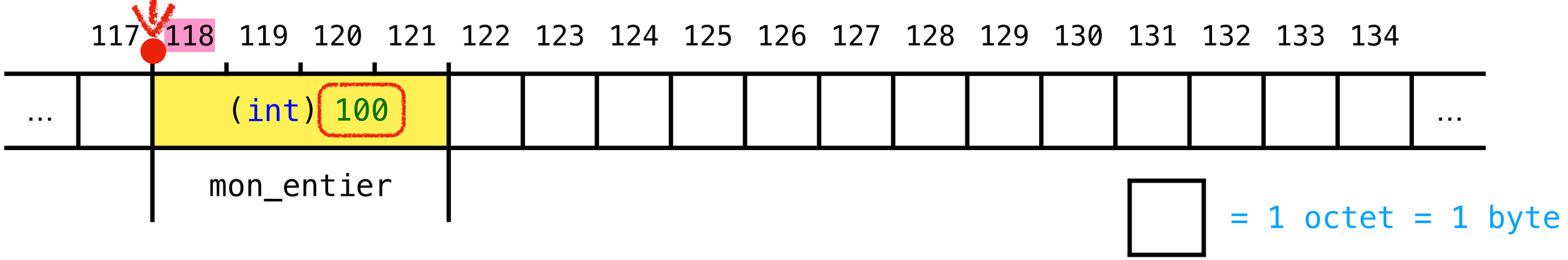


Rappels

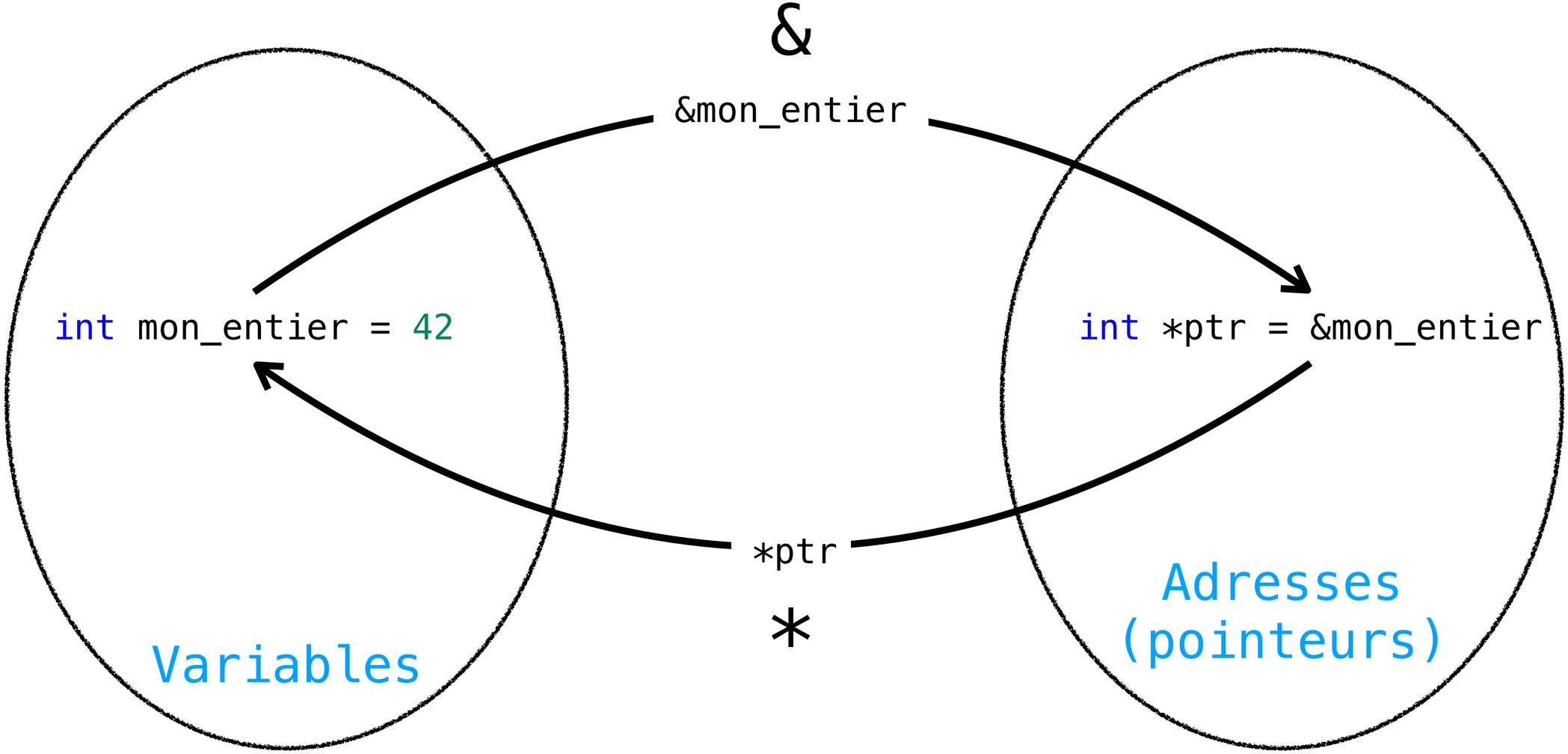
- Un pointeur vers `int` stocke une adresse vers l'emplacement d'un `int`
- Il est de type `(int *)`
- A ne pas confondre avec l'opérateur d'indirection `*ptr`

```
int *ptr = &mon_entier;
// contient 118
*ptr = 100;
```

`(*ptr)` est équivalent à `(mon_entier)`



Les opérateurs & et *

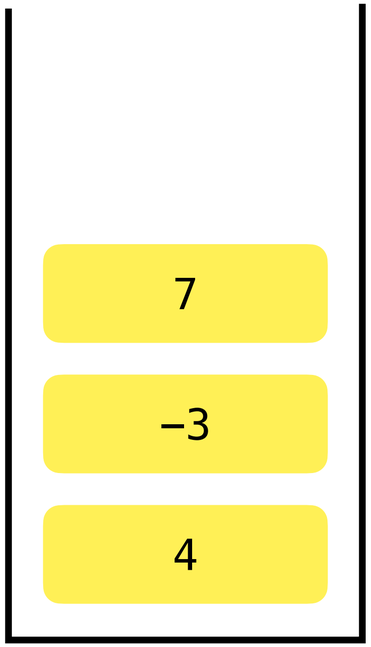


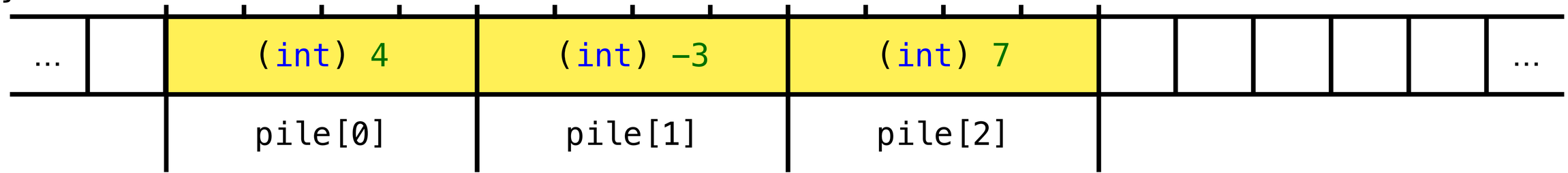
Retour sur la pile

Une implémentation possible

```c
int pile[100];
int taille, taille_max;

int push(int objet)
{
    if (taille < taille_max)
    {
        // Empiler
        pile[taille++] = objet;
        return 1;
    }
    // Pas possible, la pile est remplie
    return 0;
}
```

```c
push(4);
// taille vaut 1

push(-3);
// taille vaut 2

push(7);
// taille vaut 3
```



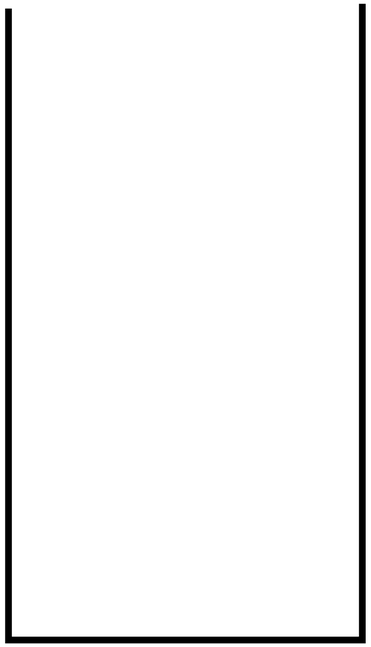


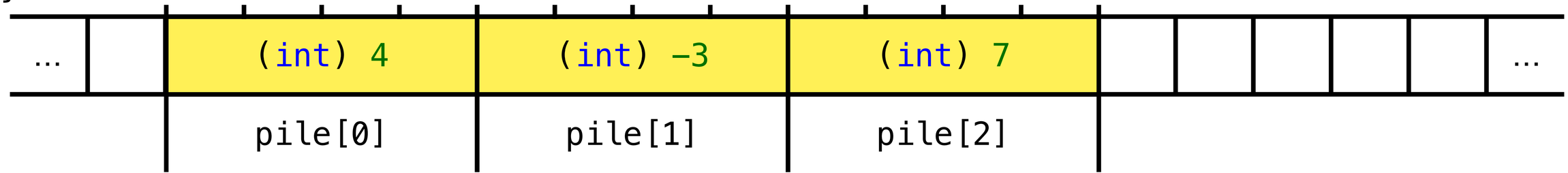
Retour sur la pile

Une implémentation possible

```c
int pile[100];
int taille, taille_max;

int pop(int *p_objet)
{
    if (taille > 0)
    {
        // Enlever du haut de la pile
        *p_objet = pile[--taille];
        return 1;
    }
    // Pas possible, la pile est vide
    return 0;
}
```

```c
int o;

pop(&o);
// o vaut 7
// taille vaut 2

pop(&o);
// o vaut -3
// taille vaut 1

pop(&o);
// o vaut 4
// taille vaut 0
```

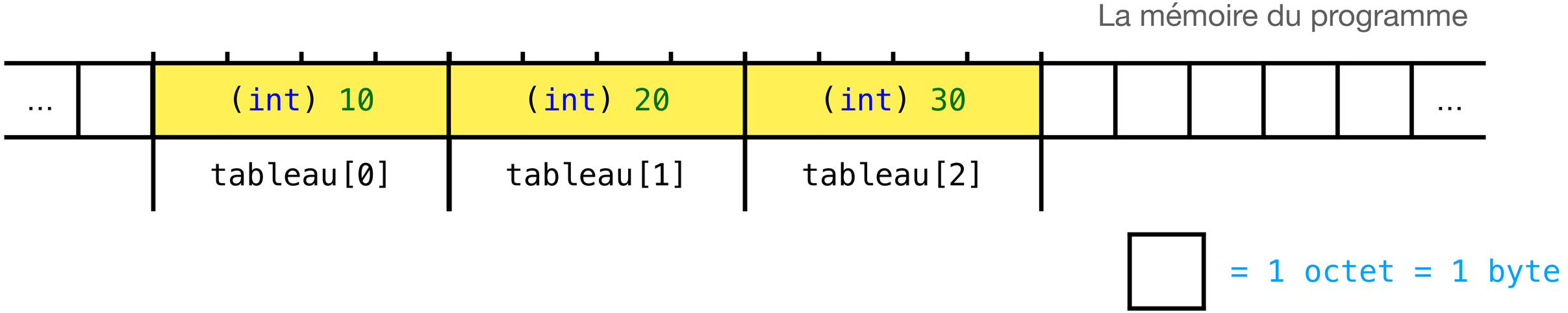




Les tableaux

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

`sizeof(tableau)` est 12



Pointeurs et tableaux

```c
void incrementer(int tableau_param[3], int taille)
{
  for (int i = 0; i < taille; i++)
    tableau_param[i]++;
}

int main()
{
  int tableau[3] = {10, 20, 30};

  incrementer(tableau, 3);
  // tableau vaut {11, 21, 31}

  return 0;
}
```

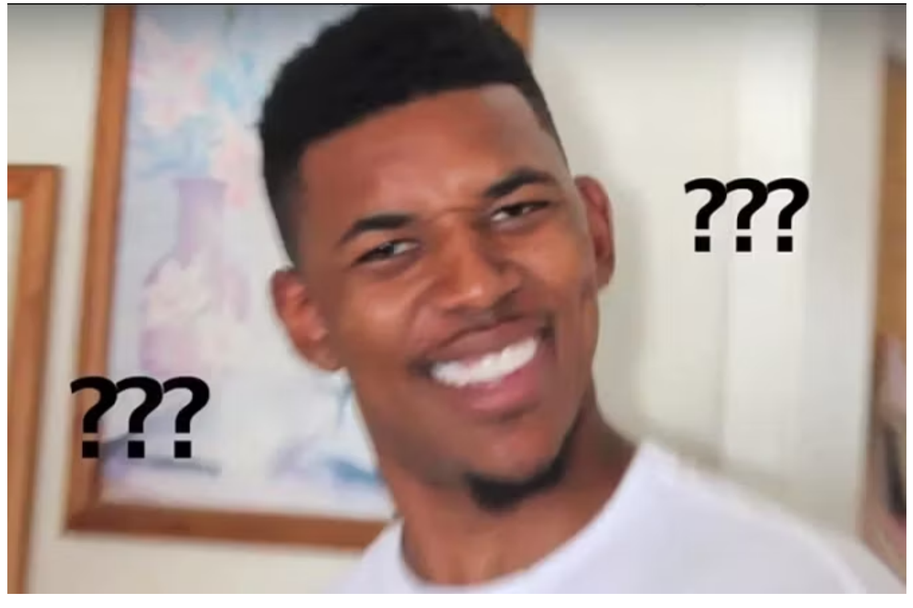
Pourquoi en modifiant le paramètre `tableau_param` modifie-t-on aussi l'argument `tableau` ?

Pointeurs et tableaux

Indice

```c
void incrementer(int tableau_param[3], int taille)
{
  printf("param: taille = %ld\n", sizeof(tableau_param));
  for (int i = 0; i < taille; i++)
    tableau_param[i]++;
}

int main()
{
  int tableau[3] = {10, 20, 30};

  printf("main: taille = %ld", sizeof(tableau));

  incrementer(tableau, 3);
  // tableau vaut {11, 21, 31}

  return 0;
}
```

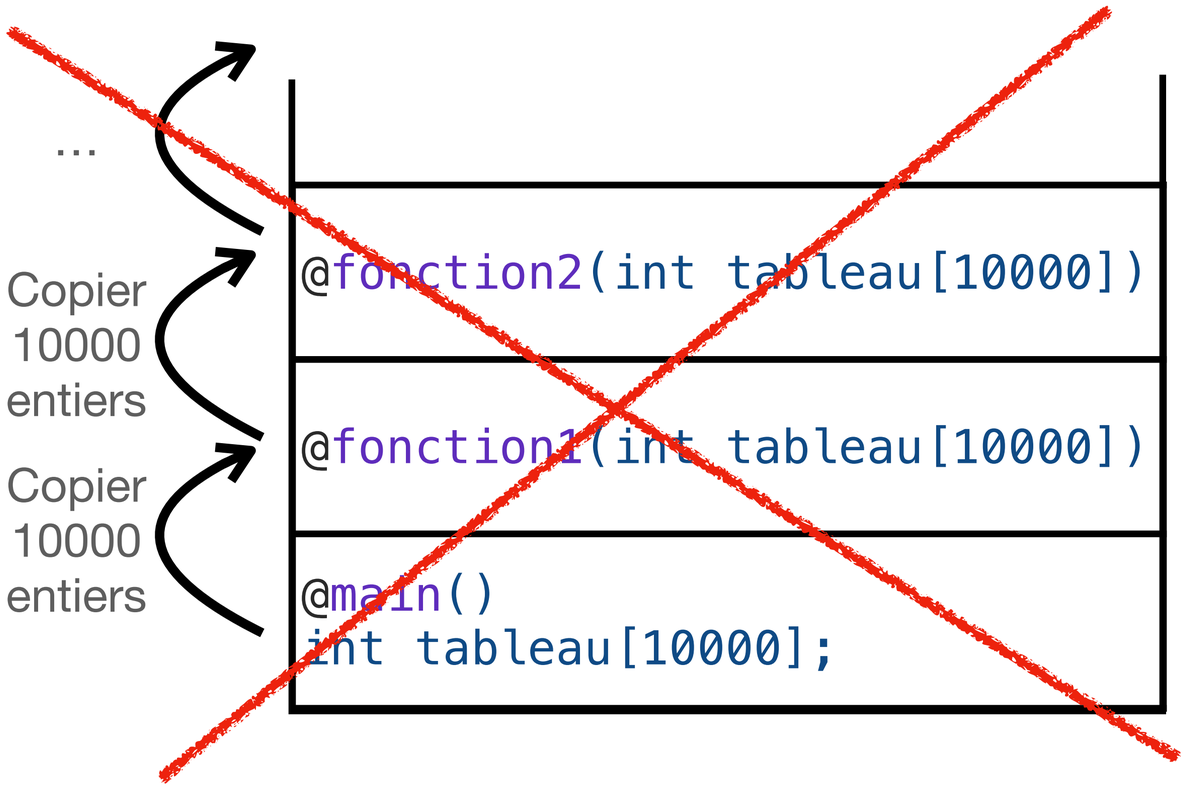
```
main: taille = 12
param: taille = 8
```



Passage des tableaux

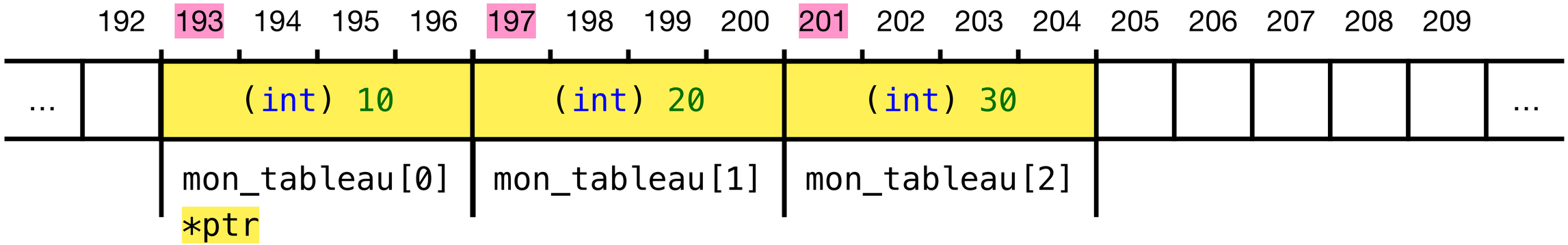
Tout paramètre de type tableau T [ ]
est implicitement converti en pointeur vers le premier élément T∗

- Pourquoi?
- Considérations pratiques…
- **Ça coûte trop cher** de copier un tableau à chaque appel
- En quelques appels la mémoire de la pile est épuisée



Pointeur vers un tableau

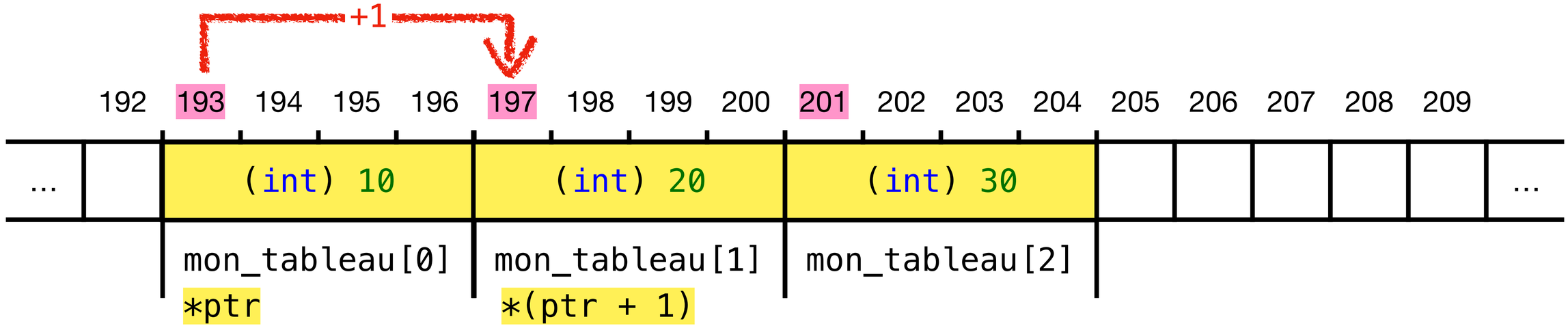
```c
int mon_tableau[3] = {10, 20, 30};
int *ptr;

ptr = mon_tableau; // conversion implicite!
                   // ptr vaut 193
                   // même chose que ptr = &mon_tableau[0]

printf("Valeur %d\n", *ptr);
// Affiche: Valeur 10
```



Bouger le pointeur?

```c
int mon_tableau[3] = {10, 20, 30};
int *ptr = mon_tableau; // vaut 193

printf("Valeur %d\n", *ptr);
// Affiche: Valeur 10

printf("Voisin %d\n", *(ptr + 1));
// Affiche: Voisin 20
```

`(ptr + 1)` signifie
"déplace l'adresse stockée dans `ptr` de `sizeof(int)` octets"
(c'est à dire 193+4 = 197)



Arithmétique des pointeurs

Somme avec des entiers

- `T *ptr` est un pointeur vers un type `T`
- `T vec[]` est un tableau qui se trouve à l'adresse `A`
- On initialise `ptr = vec`

Expression	Adresse	Emplacement équivalent
`ptr`	`A`	`&vec[0]`
`ptr + 1`	`A + sizeof(T)`	`&vec[1]`
`ptr + k`	`A + k * sizeof(T)`	`&vec[k]`

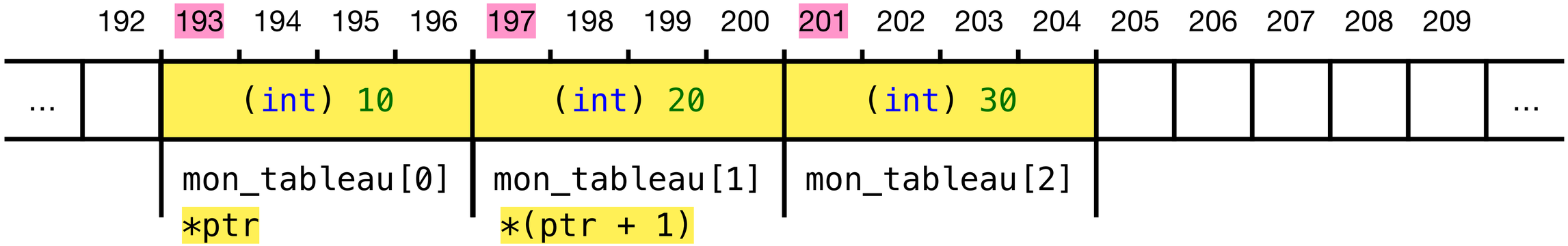
Arithmétique des pointeurs

```c
int mon_tableau[3] = {10, 20, 30};
int *ptr = mon_tableau;
```

- Ordre des opérations - * a la priorité sur +

```c
printf("Avec () %d\n", *(ptr + 1)); // Affiche: Avec () 20
printf("Sans () %d\n", *ptr + 1);   // Affiche: Sans () 11
```

- `*ptr + 1 -> (*ptr) + 1 -> 10 + 1 -> 11`



Opérateur [ ]

- C’est embêtant d’écrire `*(ptr + 3)` pour accéder au 4è élément
- 🙏 Plutôt que `*(ptr + k)` on peut tout simplement écrire `ptr[k]`
- Comme un tableau, mais pas un tableau

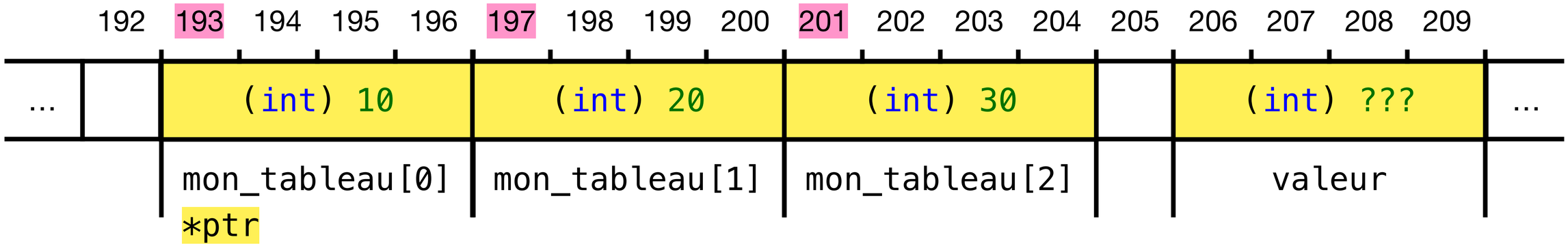
Emplacement	Adresse	Syntaxe équivalente
`*ptr`	`A`	`ptr[0]`
`*(ptr + 1)`	`A + sizeof(T)`	`ptr[1]`
`*(ptr + k)`	`A + k * sizeof(T)`	`ptr[k]`

Opérateur d’incrémentation

Qu’affiche ce code ?

```c
int mon_tableau[3] = {10, 20, 30};
int *ptr = mon_tableau;

int valeur = *ptr++;

printf("Valeur = %d\n", valeur);
printf("ptr[0] = %d\n", ptr[0]);
```

Option 1	Option 2	Option 3
Valeur = 10 ptr[0] = 11	Valeur = 10 ptr[0] = 20	Valeur = 11 ptr[0] = 193



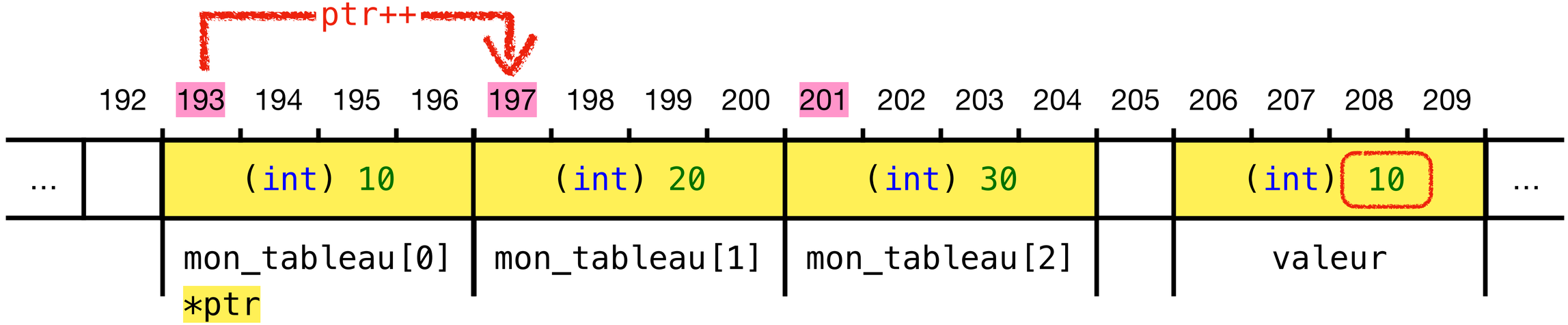
Opérateur d'incrémentation

Qu'affiche ce code ?

```c
int mon_tableau[3] = {10, 20, 30};
int *ptr = mon_tableau;

int valeur = *ptr++;

printf("Valeur = %d\n", valeur);
printf("ptr[0] = %d\n", ptr[0]);
```

⚠️ Ordre des opérations!
++ a la priorité sur *

```
valeur = *ptr++
valeur = *(ptr++)
valeur = *(193) // et ptr vaut 197

Donc le code affiche

Valeur = 10
ptr[0] = 20 // car ptr[0]
            // est la même chose que
            // *ptr
```

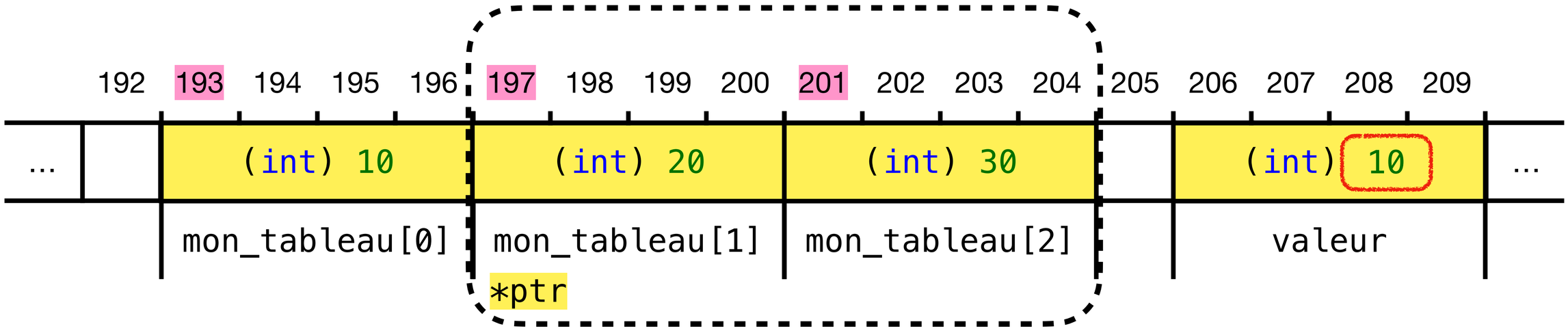


Opérateur d’incrémentation

Qu’affiche ce code ?

```c
int mon_tableau[3] = {10, 20, 30};
int *ptr = mon_tableau;

int valeur = *ptr++;

printf("Valeur = %d\n", valeur);
printf("ptr[0] = %d\n", ptr[0]);
```

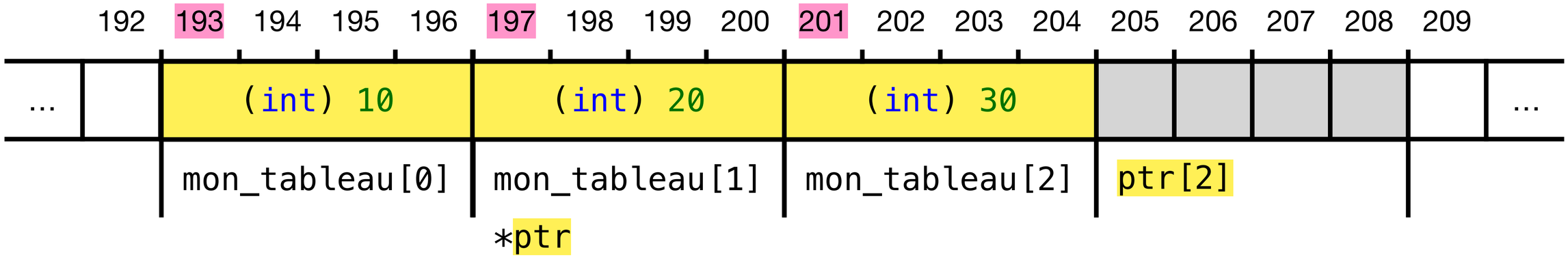
- Après l’incrémentation on peut utiliser `ptr` comme un tableau à 2 éléments
- C’est *nous* qui savons que sa taille est de 2 éléments, **pas le compilateur**



Segmentation fault

- Que se passe-t-il si on écrit dans `ptr[2]` ?
- Si l'emplacement n'est pas utilisé — ok…
- Si l'emplacement n'est pas libre: `segmentation fault`

```c
int mon_tableau[3] = {10, 20, 30};
int *ptr = mon_tableau;

int valeur = *ptr++;

printf("Valeur = %d\n", valeur);
printf("ptr[0] = %d\n", ptr[0]);

ptr[2] = 13; // dangereux!!
```



Ctrl-C Ctrl-V

- Ecrire une fonction qui prend un tableau d’entiers en paramètre (et sa taille)
- … et qui produit **une copie du tableau**
- = nouveau tableau qui contient les mêmes valeurs dans le même ordre
- Comment peut-on faire ?

Option 1

```c
int main()
{
    int tableau_original[6] =
        {7, 3, 11, 3, 1, 9};
    int taille = 6;

    printf("Tableau original: ");
    afficher(tableau_original, taille);

    int copie1[6];
    copier_statique(tableau_original,
                    taille,
                    copie1);

    printf("Copie1: ");
    afficher(copie1, taille);
}
```

```c
void copier_statique(const int *tableau,
                     int taille,
                     int *nouveau_tableau)
{
    for (int i=0; i<taille; i++)
    {
        nouveau_tableau[i] = tableau[i];
    }
}
```

```
Tableau original: 7 3 11 3 1 9
Copie1: 7 3 11 3 1 9
```

Option 2

Retourner la copie 😎

```c
int main()
{
    int tableau_original[6] =
        {7, 3, 11, 3, 1, 9};
    int taille = 6;

    printf("Tableau original: ");
    afficher(tableau_original, taille);

    int *copie2 = copier_mieux(
       tableau_original, taille);

    printf("Copie2: ");
    afficher(copie2, taille);
}
```

```c
int *copier_mieux(
    const int *tableau,
    int taille)
{
    int nouveau_tableau[taille];
    for (int i = 0; i < taille; i++)
    {
        nouveau_tableau[i] = tableau[i];
    }
    return nouveau_tableau;
}
```

```
Tableau original: 7 3 11 3 1 9
Copie2: 1863872528 1 1863872752 1 13090396 1
```

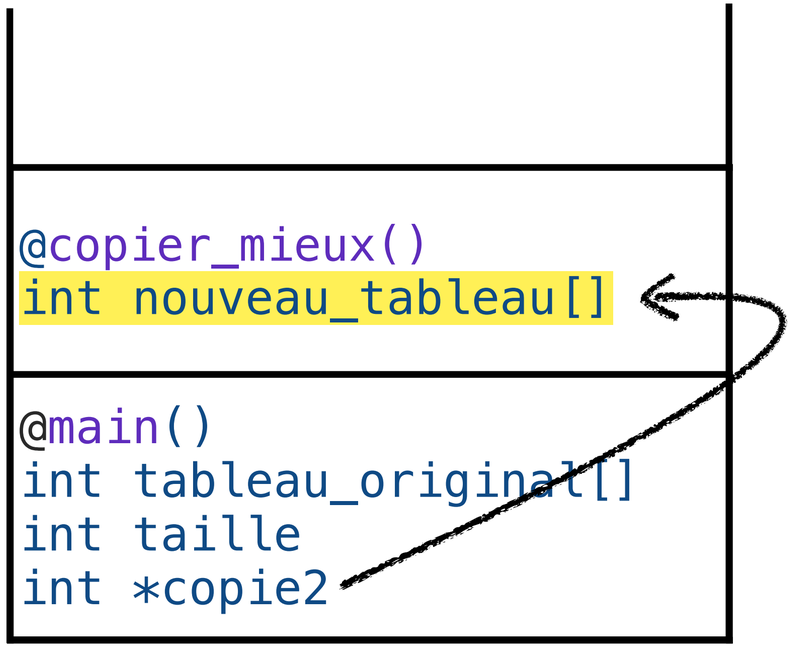
Pourquoi?! 😩

“Le mieux est l’ennemi du bien”

- Indice: le compilateur nous donne l’avertissement suivant:

```
tri.c:27:12: warning: address of stack memory associated with
local variable 'nouveau_tableau' returned [-Wreturn-stack-
address]
    return nouveau_tableau;
           ^~~~~~~~~~~~~~~
```

- Les variables locales n’existent **que pendant l’appel** de la fonction
- Quand l’appel est fini, les variables locales sont **détruites**
- Au retour `copie2` pointe vers l’ancien emplacement de `nouveau_tableau` qui est devenu **invalide**



A new hope

- Nous pouvons demander des emplacements de mémoire **hors de la pile**!
- C’est une requête **explicite** qu’on fait au système d’exploitation
- Allocation dynamique de la mémoire
- On reçoit un pointeur vers un emplacement de mémoire qui sera réservé jusqu’à la fin de l’exécution du programme
- L’emplacement de mémoire se trouvera dans une région de la mémoire qui s’appelle “le tas” (*the heap*)

`malloc` et `free`
`#include <stdlib.h>`

- `malloc` (*Memory allocation*)
  - Prend comme unique paramètre **le nombre d'octets** à réserver
  - **Retourne:** un pointeur vers l'emplacement de mémoire réservé sur le tas
  - En cas d'échec — retourne le **pointeur nul**, i.e., `NULL`
  - Le type de retour est `(void *)` — pointeur générique sans type
  - ⚠️ Il faut allouer la bonne quantité de mémoire et (si besoin) convertir vers le bon type
  - Par exemple, `(float *) malloc(10 * sizeof(float)); // 10 x float`

`malloc` et `free`

- `free` (Libérer la mémoire)
  - Prend comme unique paramètre **le pointeur** vers l'emplacement à libérer
  - **Retourne** : rien
- **Toujours** libérer, délivrer la mémoire!

Option 3

Copie dynamique

```c
int main()
{
    int tableau_original[6] =
        {7, 3, 11, 3, 1, 9};
    int taille = 6;

    printf("Tableau original: ");
    afficher(tableau_original, taille);

    int *copie3 = copier_dynamique(
        tableau_original,
        taille);

    printf("Copie3: ");
    afficher(copie3, taille);

    free(copie3);

}
```

```c
int *copier_dynamique(
    const int *tableau,
    int taille)
{
    int *nouveau_tableau =
         malloc(taille * sizeof(int));
    for (int i = 0; i < taille; i++)
    {
        nouveau_tableau[i] = tableau[i];
    }
    return nouveau_tableau;
}
```

```
Tableau original: 7 3 11 3 1 9
Copie3: 7 3 11 3 1 9
```