

Cours 7

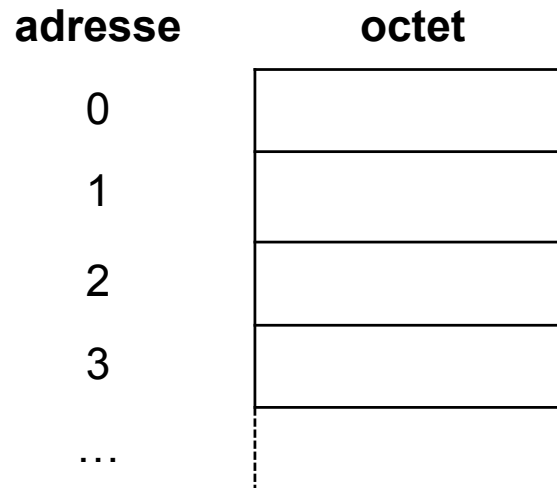
Variables, tableaux, fonctions et mémoire

Vincent Lepetit

`vincent.lepetit@epfl.ch`

Mémoire

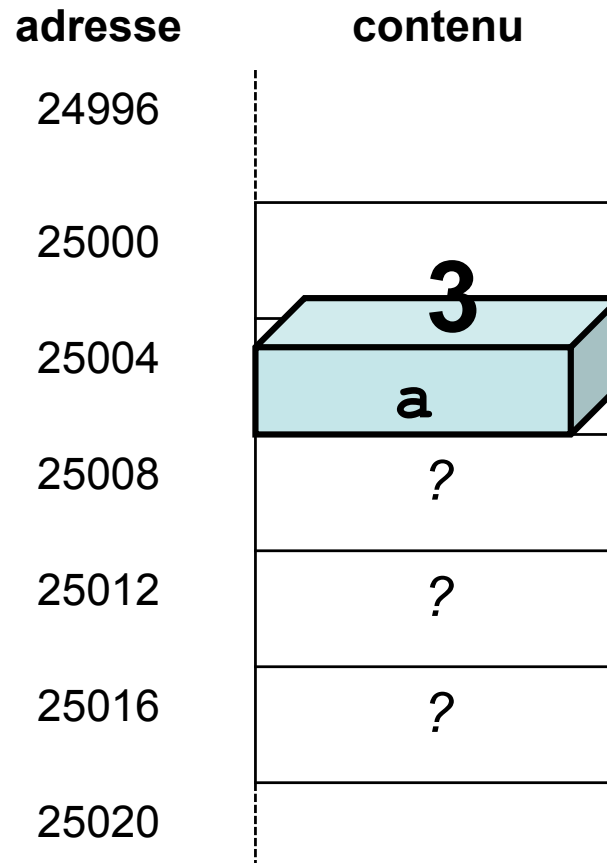
La mémoire d'un ordinateur est composée d'un grand nombre **d'octets** (une valeur entre 0 et 255) :



Chaque octet est repéré par une **adresse**, qui est **un nombre entier**.

Les variables sont stockées en mémoire. Par exemple:

```
int a = 3;
```



L'opérateur &

On peut obtenir l'adresse d'une variable grâce à l'opérateur & :

Par exemple:

```
int a = 3;
```

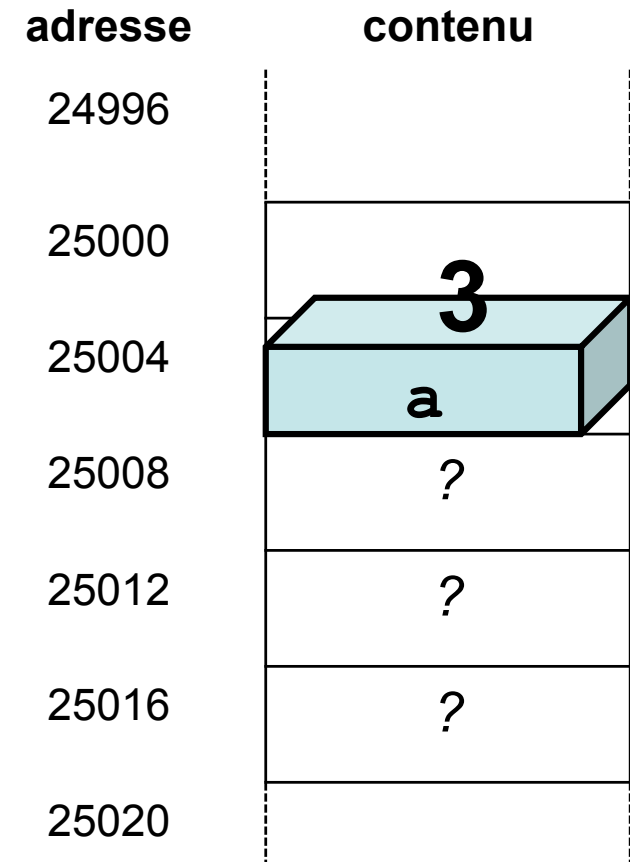
```
cout << a << endl;
```

```
cout << (unsigned int) (&a) << endl;
```

va afficher pour cet exemple:

3

25004



Stocker une adresse dans une variable

En C et en C++, on peut définir des variables pour qu'elles contiennent des adresses.

Pour déclarer une telle variable, Il faut ajouter une étoile * entre le type et le nom de la variable:

```
int * p;
```

p est appelé un *pointeur*.

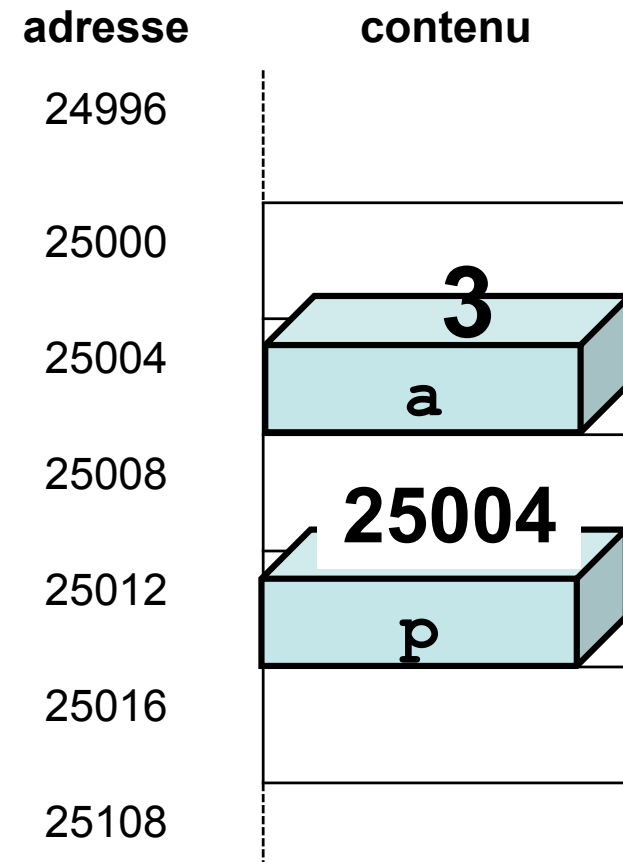
Stocker une adresse dans une variable

Si on fait:

```
int * p;
```

```
p = &a;
```

p contient l'adresse de a.



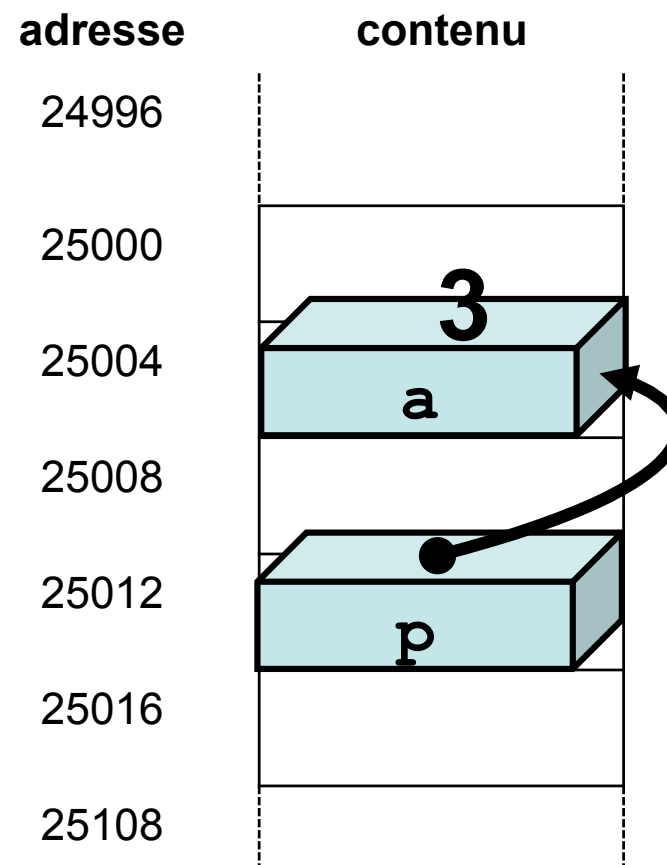
Représentation graphique (1)

p contient l'adresse de a .

On dit que p pointe sur a .

La valeur de l'adresse de a n'est pas importante en elle-même, ce qui est important est que p pointe sur a .

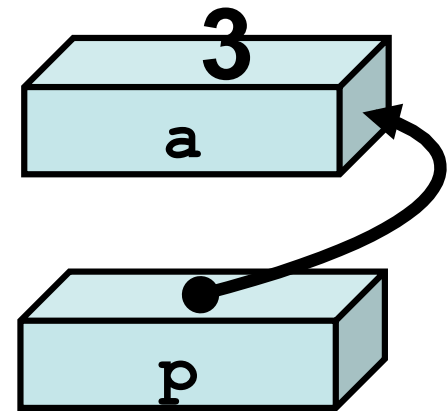
Ceci est généralement représenté par une flèche allant de p à a .



Représentation graphique (2)

On n'a généralement pas besoin de la représentation de la mémoire, et on peut ne garder que la représentation des variables.

```
int a = 3;  
int * p;  
p = &a;
```

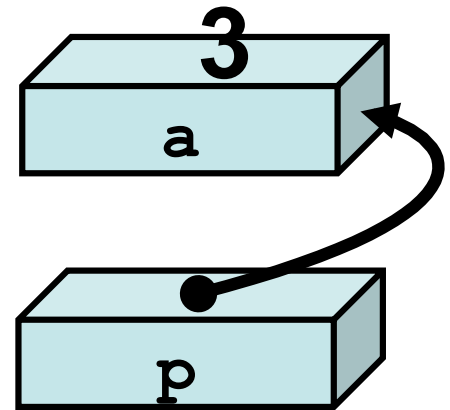


L'opérateur *

L'opérateur * permet d'accéder à la valeur stockée à une adresse:

```
cout << *p << endl;
```

affiche 3.



Modifier la valeur pointée par un pointeur

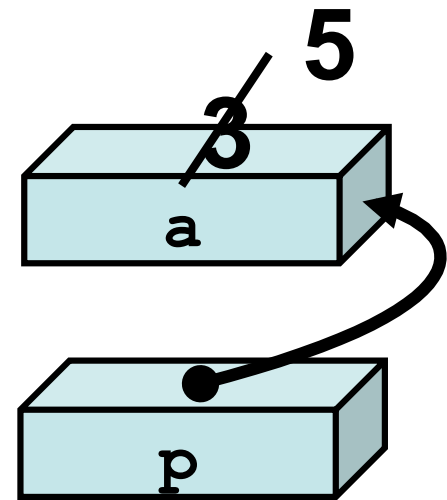
On peut aussi utiliser l'opérateur pour modifier une valeur pointée par un pointeur:

```
int a = 3;
```

```
int * p;
```

```
p = &a;
```

```
*p = 5;
```



Attention à ne pas confondre

- l'étoile utilisée pour déclarer un pointeur:

`int *p;`

Même symbole (*) mais
signification différente.

- et l'étoile pour obtenir la valeur pointée par le pointeur:

`*p = 5;`

Résumé

Un pointeur est une variable. On déclare un pointeur en mettant une étoile (*) entre le type et le nom du pointeur:

```
int * p;
```

Un pointeur contient une adresse.

On peut obtenir l'adresse d'une variable avec l'opérateur &:

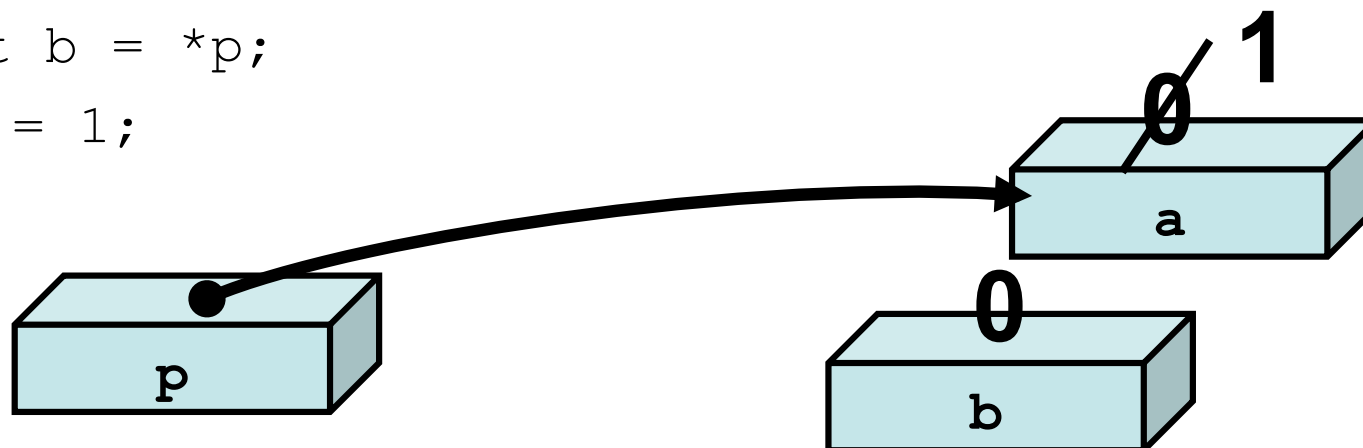
```
int a = 0;
```

```
p = &a;
```

L'opérateur * permet d'accéder à la mémoire pointée par un pointeur:

```
int b = *p;
```

```
*p = 1;
```



Déclaration d'un pointeur

Un pointeur est une variable:

On peut déclarer plusieurs pointeurs sur une même ligne, et initialiser un pointeur lors de sa déclaration. Exemples:

```
float * p1 = &a;  
int * p2, * p3;
```

Attention: pour déclarer plusieurs pointeurs sur une même ligne, il faut *répéter* l'étoile. Si on fait:

```
int * p2, p3; // !!
```

`p2` est de type pointeur sur `int`, mais `p3` est une variable classique, de type `int`.

Le pointeur NULL ou 0

On utilise parfois la valeur 0, ou la constante `NULL` (de type pointeur, qui vaut 0) pour initialiser un pointeur:

```
int * p = 0;
```

ou

```
int * p = NULL;
```

L'adresse 0 n'est jamais utilisée pour stocker des variables, elle correspond donc toujours à une adresse invalide: Si on essaie d'accéder à la valeur pointée par `p` en faisant par exemple:

```
int a = *p;
```

```
*p = 0;
```

on provoque un:

```
Segmentation fault
```

L'intérêt du pointeur `NULL` est par exemple de signaler qu'une fonction n'a pas pu fonctionner correctement, ou qu'un pointeur ne contient pas une adresse valide.

Remarques

- On peut voir l'opérateur `*` comme l'inverse de l'opérateur `&`:

```
b = *(&a);
```

est équivalent à

```
b = a;
```

- On peut mettre zéro, un ou plusieurs espaces avant et après l'étoile et le *et commercial*:

```
int *pa = & a;
```

```
b = * pa;
```

Affectation de pointeurs

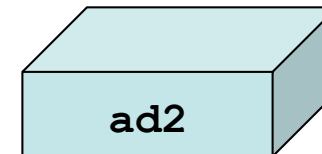
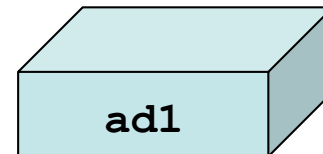
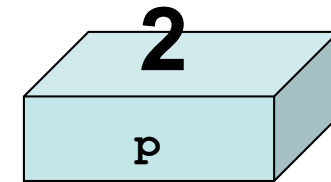
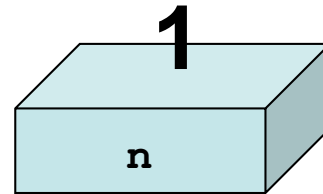
On peut affecter à un pointeur la valeur d'un autre pointeur de même type.

Exemple:

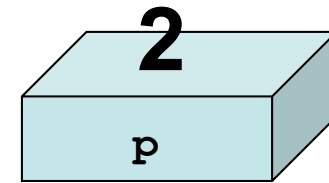
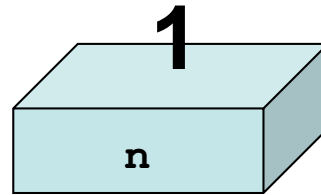
```
int n = 1, p = 2;  
int * ad1, * ad2;
```

Que font les instructions:

```
ad1 = &n;  
ad2 = &p;  
*ad1 = *ad2 + 3;  
ad1 = ad2;  
*ad1 = *ad2 + 5;
```



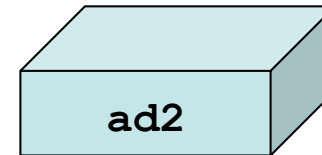
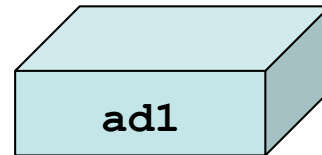
Pas-à-pas



```
int n = 1, p = 2;  
int * ad1, * ad2;
```



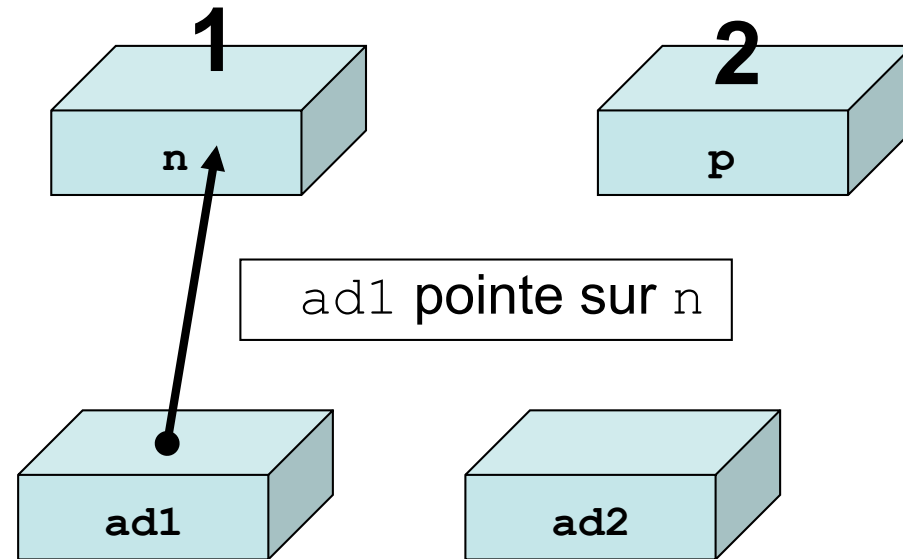
```
ad1 = &n;  
ad2 = &p;  
*ad1 = *ad2 + 3;  
ad1 = ad2;  
*ad1 = *ad2 + 5;
```



Pas-à-pas

```
int n = 1, p = 2;  
int * ad1, * ad2;
```

```
→ ad1 = &n;  
   ad2 = &p;  
   *ad1 = *ad2 + 3;  
   ad1 = ad2;  
   *ad1 = *ad2 + 5;
```



Pas-à-pas

```
int n = 1, p = 2;  
int * ad1, * ad2;
```

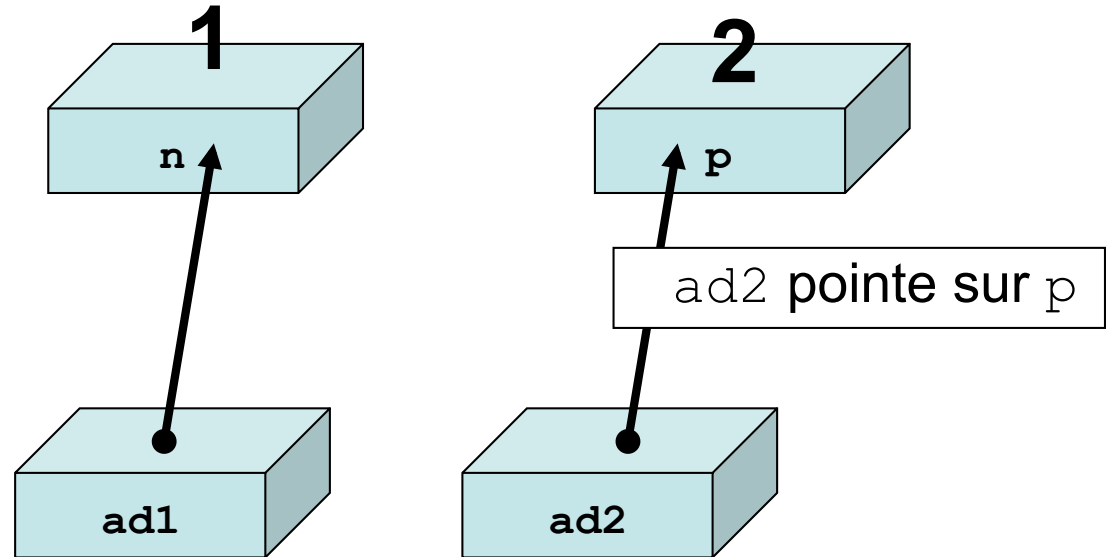
```
ad1 = &n;
```

```
→ ad2 = &p;
```

```
*ad1 = *ad2 + 3;
```

```
ad1 = ad2;
```

```
*ad1 = *ad2 + 5;
```



Pas-à-pas

```
int n = 1, p = 2;  
int * ad1, * ad2;
```

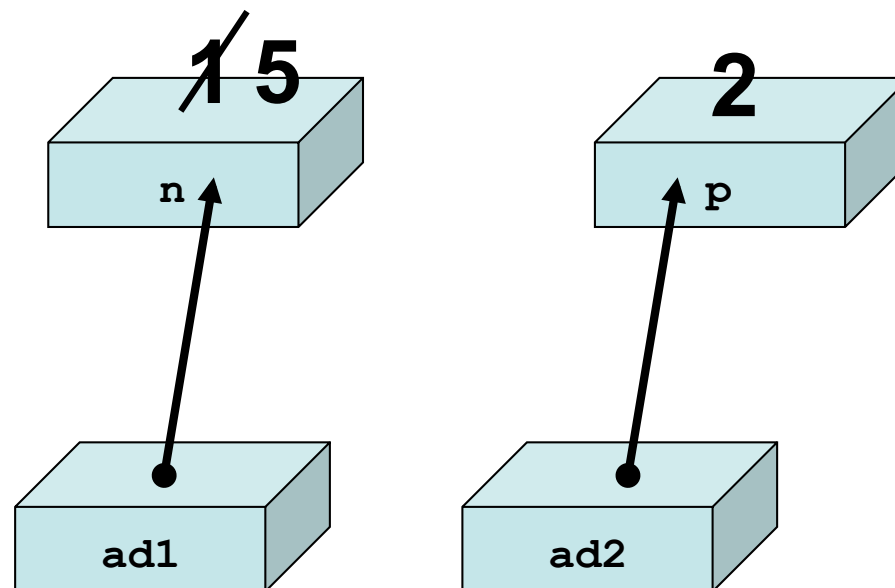
```
ad1 = &n;
```

```
ad2 = &p;
```

```
→ *ad1 = *ad2 + 3;
```

```
ad1 = ad2;
```

```
*ad1 = *ad2 + 5;
```



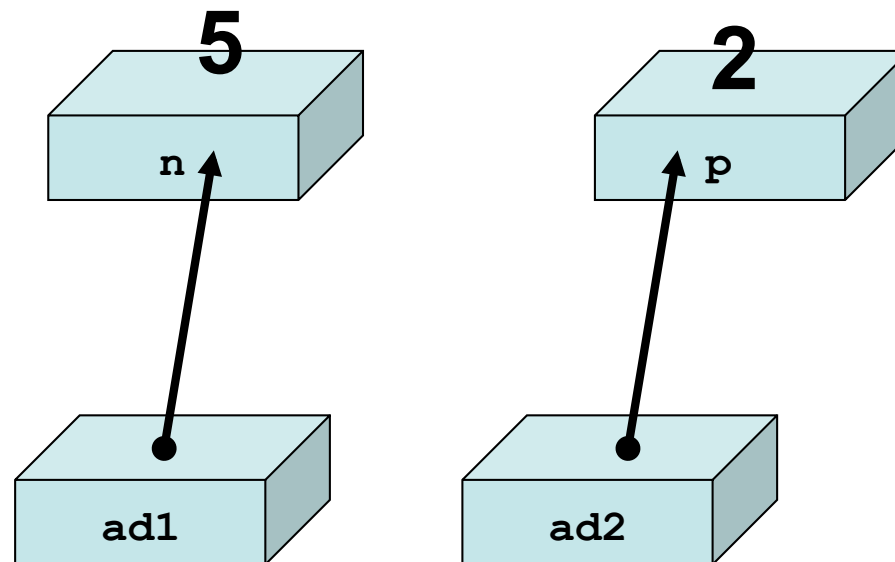
exactement comme $n = p + 3$;
n contient maintenant 5

Pas-à-pas

```
int n = 1, p = 2;  
int * ad1, * ad2;
```

```
ad1 = &n;  
ad2 = &p;  
*ad1 = *ad2 + 3;
```

```
→ ad1 = ad2;  
*ad1 = *ad2 + 5;
```

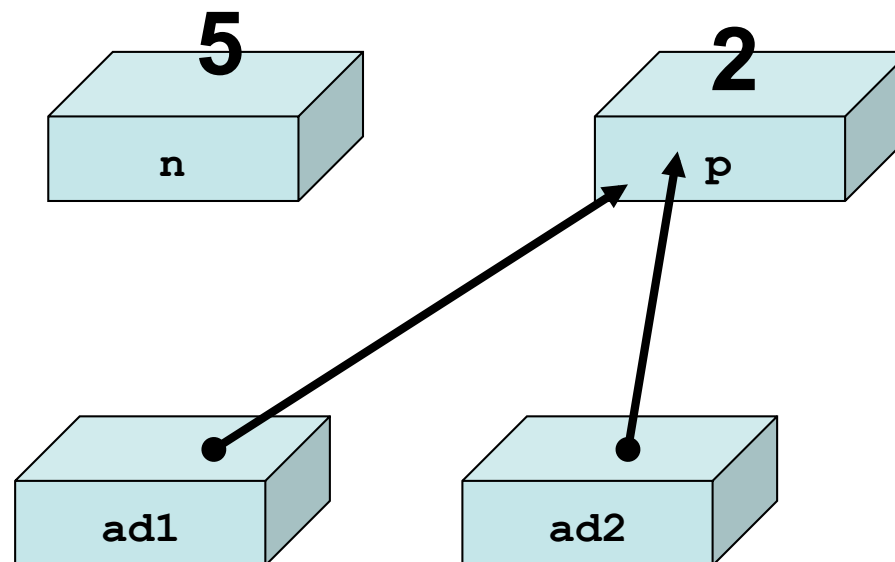


Pas-à-pas

```
int n = 1, p = 2;  
int * ad1, * ad2;
```

```
ad1 = &n;  
ad2 = &p;  
*ad1 = *ad2 + 3;
```

```
→ ad1 = ad2;  
*ad1 = *ad2 + 5;
```



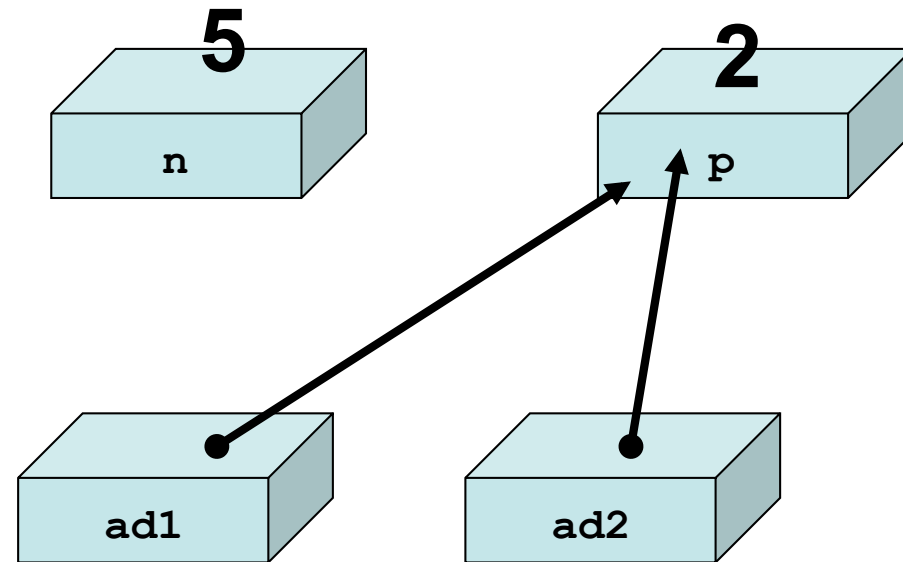
ad1 et ad2 pointent maintenant tous les deux sur p.

Pas-à-pas

```
int n = 1, p = 2;  
int * ad1, * ad2;
```

```
ad1 = &n;  
ad2 = &p;  
*ad1 = *ad2 + 3;  
ad1 = ad2;
```

→ *ad1 = *ad2 + 5;



exactement comme $p = p + 5$;
 p contient donc maintenant 7

Exercise

```
int a = 2, b = 5;
```

```
int * p;
```

```
p = &a;
```

```
cout << a << " " << *p << endl;
```

```
a = a + 1;
```

```
cout << a << " " << *p << endl;
```

```
*p = b;
```

```
cout << a << " " << *p << endl;
```

```
b = b + 1;
```

```
cout << a << " " << *p << endl;
```

Première application

Fonction modifiant une variable
passée en paramètre

Rappel

Si on exécute:

```
void mis_a_zero(int a)
{
    a = 0;
}

int main(int argc, char ** argv)
{
    int n = 10;

    mis_a_zero(n);

    cout << "apres appel: n = " << n << endl;
}
```

on obtient:

```
apres appel: n = 10
```

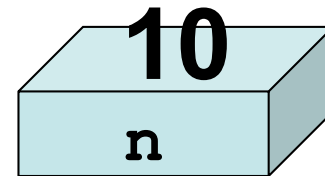
La fonction `mis_a_zero` ne peut pas changer la valeur de `n` !

Pas-à-pas

```
void mis_a_zero(int a)
{
    cout << "avant: a = " << a << endl;
    a = 0;
    cout << "apres: a = " << a << endl;
}

int main(int argc, char ** argv)
{
    int n = 10;

    cout << "avant appel: n = " << n << endl;
    → mis_a_zero(n);
    cout << "apres appel: n = " << n << endl;
}
```



Pas-à-pas

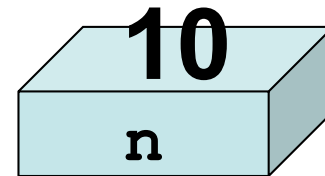
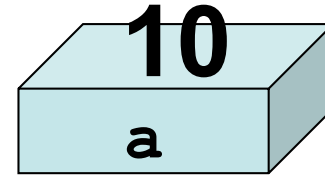
```
→ void mis_a_zero(int a)
{
    cout << "avant: a = " << a << endl;
    a = 0;
    cout << "apres: a = " << a << endl;
}

int main(int argc, char ** argv)
{
    int n = 10;

    cout << "avant appel: n = " << n << endl;

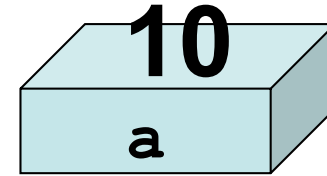
    → mis_a_zero(n);

    cout << "apres appel: n = " << n << endl;
}
```



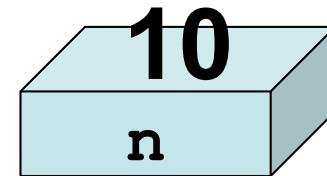
Pas-à-pas

```
void mis_a_zero(int a)
{
    cout << "avant: a = " << a << endl;
    → a = 0;
    cout << "apres: a = " << a << endl;
}
```



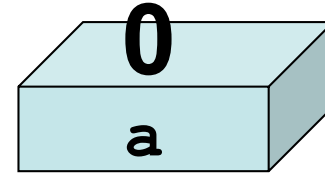
```
int main(int argc, char ** argv)
{
    int n = 10;

    cout << "avant appel: n = " << n << endl;
    → mis_a_zero(n);
    cout << "apres appel: n = " << n << endl;
}
```



Pas-à-pas

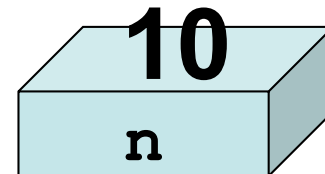
```
void mis_a_zero(int a)
{
    cout << "avant: a = " << a << endl;
    → a = 0;
    cout << "apres: a = " << a << endl;
}
```



```
int main(int argc, char ** argv)
{
    int n = 10;

    cout << "avant appel: n = " << n << endl;
    → mis_a_zero(n);

    cout << "apres appel: n = " << n << endl;
}
```



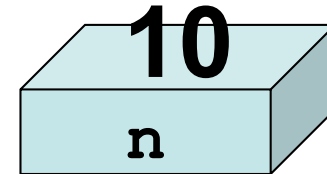
Pas-à-pas

```
void mis_a_zero(int a)
{
    cout << "avant: a = " << a << endl;
    a = 0;
    cout << "apres: a = " << a << endl;
}
```

```
int main(int argc, char ** argv)
{
    int n = 10;

    cout << "avant appel: n = " << n << endl;

    mis_a_zero(n);
    cout << "apres appel: n = " << n << endl;
}
```



Même si la fonction a changé les valeurs du paramètre formel `a`, le changement n'est pas répercuté sur le paramètre effectif `n`.

Une fonction modifiant une variable passée en paramètre

Pour que la fonction puisse modifier la variable `n`, il faut passer en paramètre l'adresse de la variable plutôt que sa valeur:

```
void mis_a_zero(int * pa)
{
    *pa = 0;
}

int main(int argc, char ** argv)
{
    int n = 10;

    mis_a_zero(&n);

    cout << "apres appel: n = " << n << endl;
}
```

on obtient cette fois:

```
apres appel: n = 0
```

Pas-à-pas

```
void mis_a_zero(int * pa)
{
    *pa = 0;
}
```

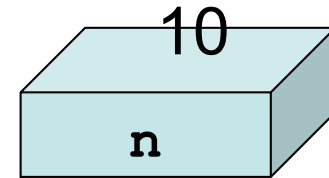
```
int main(int argc, char ** argv)
{
```

→ `int n = 10;`

```
    mis_a_zero(&n);
```

```
    cout << "apres appel: n = " << n << endl;
```

```
}
```

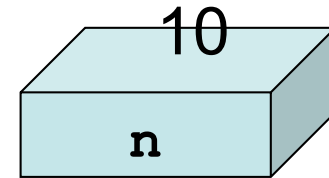


1. La variable `n` est créée en mémoire

Pas-à-pas

```
void mis_a_zero(int * pa)
{
    *pa = 0;
}
```

```
int main(int argc, char ** argv)
{
    int n = 10;
```



```
→ mis_a_zero(&n);
```

2. L'adresse de la variable `n` est passée en paramètre.

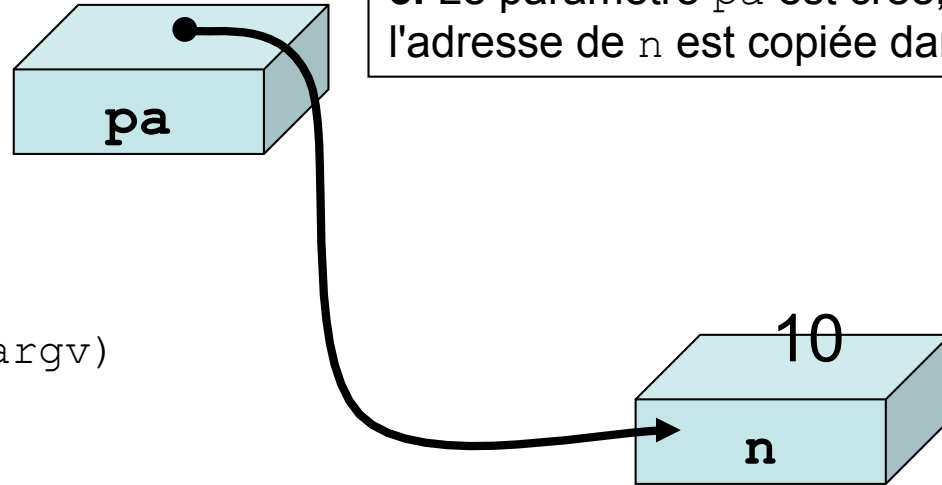
```
    cout << "apres appel: n = " << n << endl;
}
```

Pas-à-pas

3. Le paramètre `pa` est créé, l'adresse de `n` est copiée dans `pa`.

```
→ void mis_a_zero(int * pa)  
{  
    *pa = 0;  
}
```

```
int main(int argc, char ** argv)  
{  
    int n = 10;  
  
    mis_a_zero(&n);  
  
    cout << "apres appel: n = " << n << endl;  
}
```



Pas-à-pas

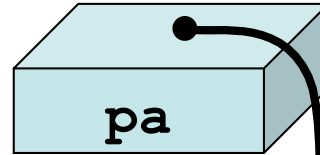
```
void mis_a_zero(int * pa)
{
  *pa = 0;
}
```



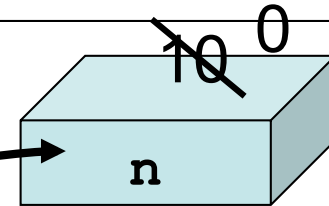
```
int main(int argc, char ** argv)
{
  int n = 10;

  mis_a_zero(&n);

  cout << "apres appel: n = " << n << endl;
}
```



4. On met 0 dans la mémoire pointée par `pa`.
Comme `pa` pointe sur `n`, on met 0 dans `n`.



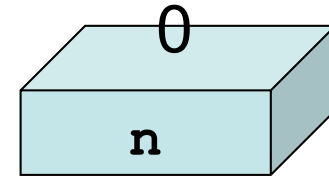
Pas-à-pas

```
void mis_a_zero(int * pa)
{
    *pa = 0;
}

int main(int argc, char ** argv)
{
    int n = 10;

    mis_a_zero(&n);

    cout << "apres appel: n = " << n << endl;
}
```



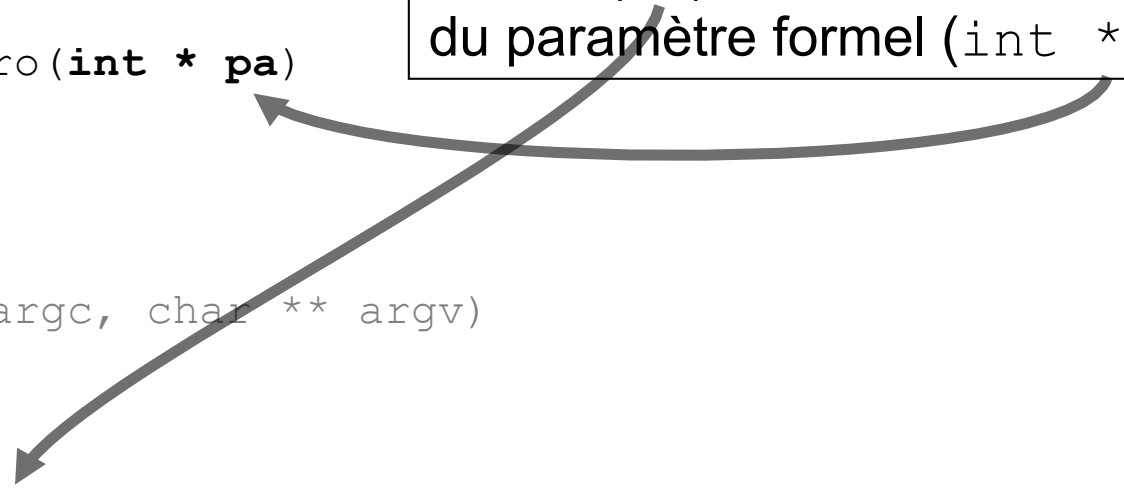
5. Après l'appel à la fonction, `n` a bien été modifiée.

On peut vérifier que le type du paramètre effectif (&n) est bien le même que celui du paramètre formel (int * pa).

```
void mis_a_zero(int * pa)
{
    *pa = 0;
}

int main(int argc, char ** argv)
{
    int n = 10;
    mis_a_zero(&n);

    cout << "apres appel: n = " << n << endl;
}
```

A diagram consisting of two curved arrows. The first arrow starts from the text 'int * pa' in the function signature and points to the parameter '&n' in the function call. The second arrow starts from the text '&n' in the function call and points to the parameter 'pa' in the function signature. This visualizes the matching of the actual parameter's type (int*) with the formal parameter's type (int*).

Recette

1. Ajouter une étoile (*) quand la fonction doit modifier la variable passée en paramètre:

```
void mis_a_zero(int * pa)
```

pa est maintenant un pointeur sur la variable passée en paramètre.

2. Dans le code de la fonction, on peut accéder à la variable passée en paramètre en ajoutant une étoile devant le nom du paramètre:

```
*pa = 0;
```

3. La fonction attend maintenant un paramètre de type pointeur. A l'appel de la fonction, il faut ajouter un & commercial devant la variable passée en paramètre:

```
mis_a_zero(&n);
```

MAIS ATTENTION...

...Attention à l'appel

Supposons que l'on ait une fonction qui ajoute 1 à une variable passée en paramètre:

```
void ajoutel(int * pn)
{
    *pn = *pn + 1;
}
```

et qu'on veuille utiliser cette fonction pour écrire une fonction qui ajoute 2 à une variable passée en paramètre.

...Attention à l'appel

La recette précédente ne marche pas dans ce cas.

La fonction `ajoute2` appelle deux fois la fonction `ajoute1` pour ajouter 2 à la variable passée en paramètre.

Mais il faut faire attention à l'expression exacte du paramètre. La version ci-dessous ne marche pas:

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    ajoute1(&pm);
    ajoute1(&pm);
}
```

`pm` contient déjà l'adresse de la variable à modifier.

`&pm` correspond à l'adresse de `pm` !

...

```
int a = 0;
```

```
ajoute2(&a);
```

...Attention à l'appel

Dans ce cas, il faut faire:

```
void ajoutel(int * pn)
{
    *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    ajoutel(pm);
    ajoutel(pm);
}
```

← pm contient déjà l'adresse de la variable à modifier.

```
...
int a = 0;

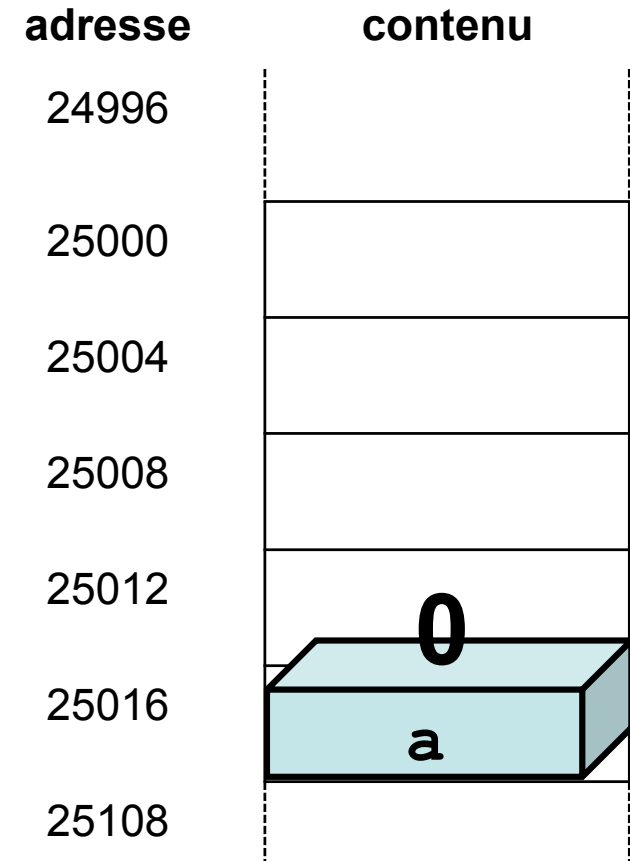
ajoute2(&a);
```

Pas-à-pas (version incorrecte)

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}

void ajoute2(int * pm)
{
    ajoute1(&pm); // faux !
    ajoute1(&pm); // faux !
}

...
int a = 0;
→ ajoute2(&a);
```

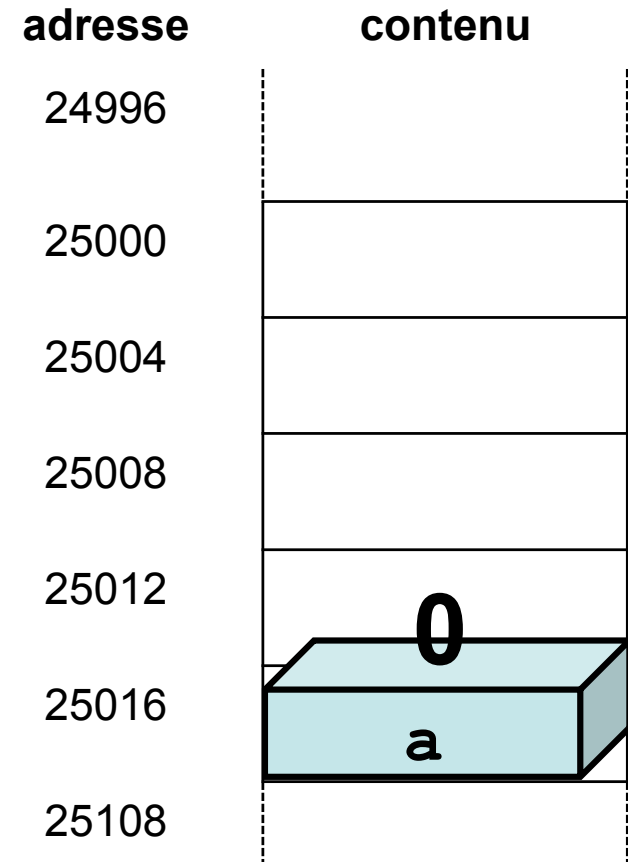


Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}

void ajoute2(int * pm)
{
    ajoute1(&pm); // faux !
    ajoute1(&pm); // faux !
}

...
int a = 0;
→ ajoute2(&a);
```



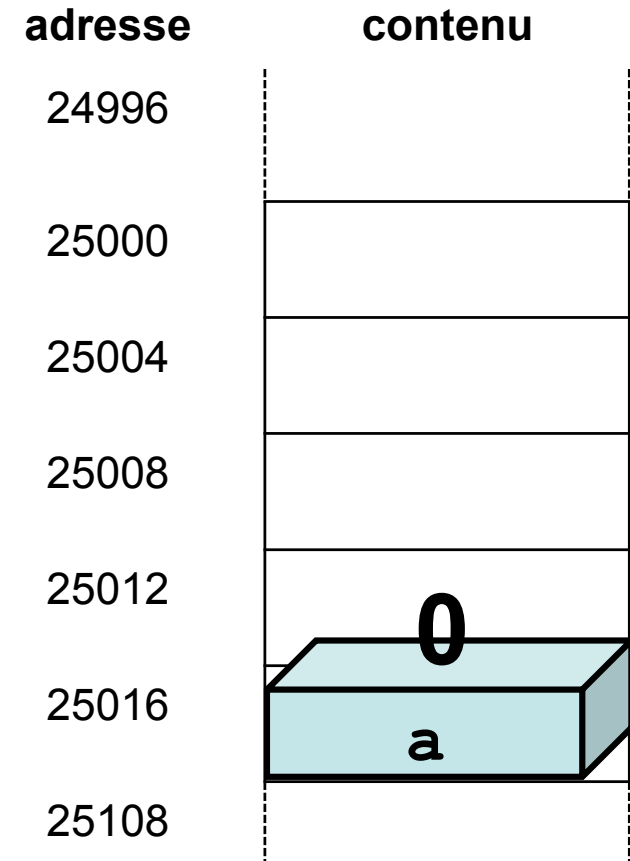
Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

```
→ void ajoute2(int * pm)
{
    ajoute1(&pm); // faux !
    ajoute1(&pm); // faux !
}
```

```
...
int a = 0;
```

```
→ ajoute2(&a);
```



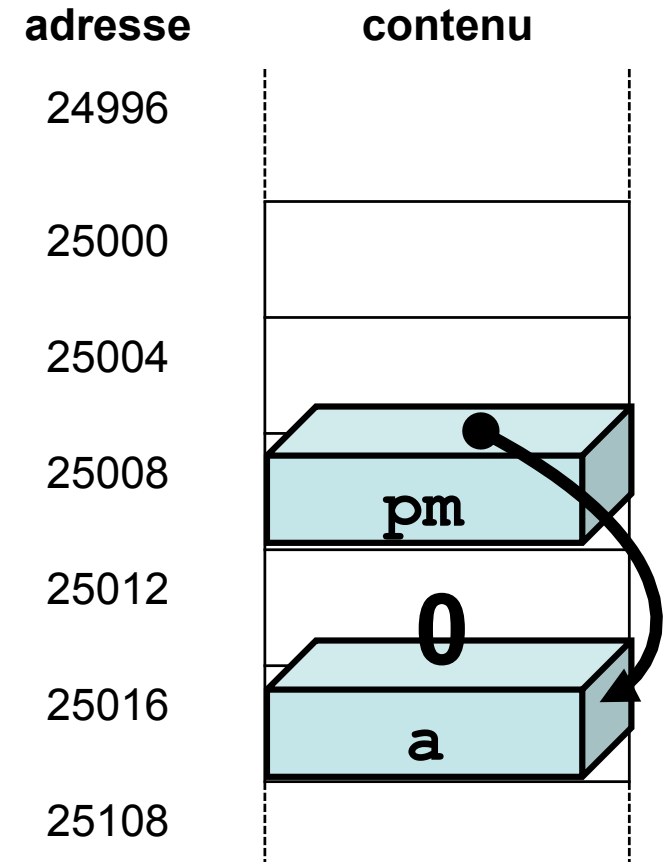
Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

```
→ void ajoute2(int * pm)
{
    ajoute1(&pm); // faux !
    ajoute1(&pm); // faux !
}
```

```
...
int a = 0;
```

```
→ ajoute2(&a);
```



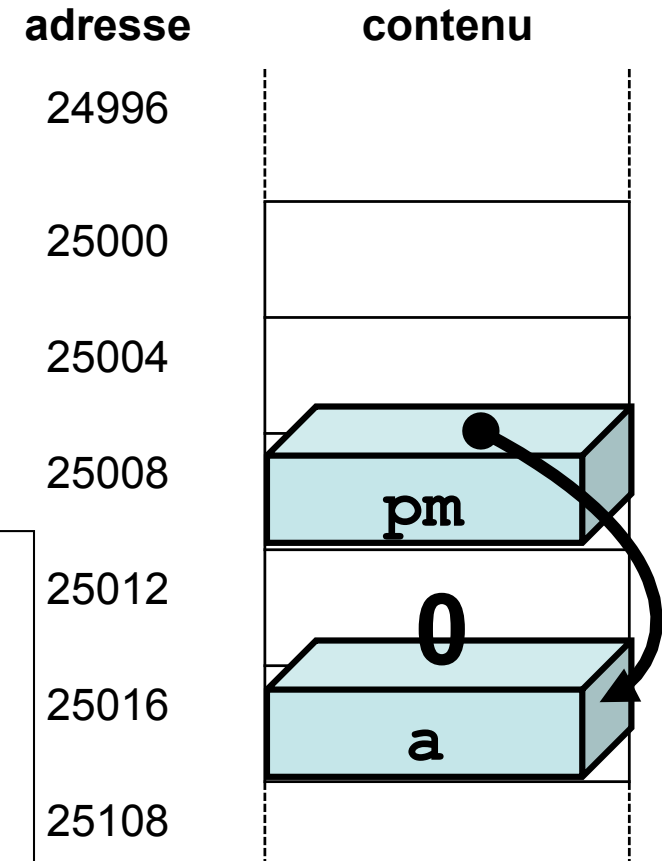
Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}

void ajoute2(int * pm)
{
    ajoute1(&pm); // faux !
    ajoute1(&pm); // faux !
}
```

&pm est l'adresse de pm, et non pas l'adresse de a.

De plus, cette instruction ne compilera pas, car la fonction ajoute1 attend un paramètre de type int * alors que &pm est de type int ** !

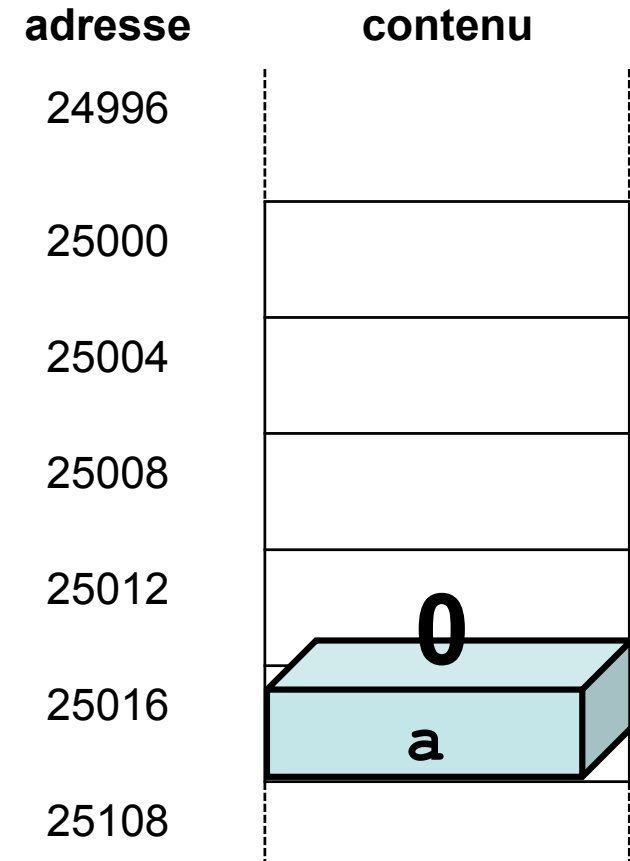


Pas-à-pas (version correcte)

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    ajoute1(pm);
    ajoute1(pm);
}
```

```
...
int a = 0;
→ ajoute2(&a);
```



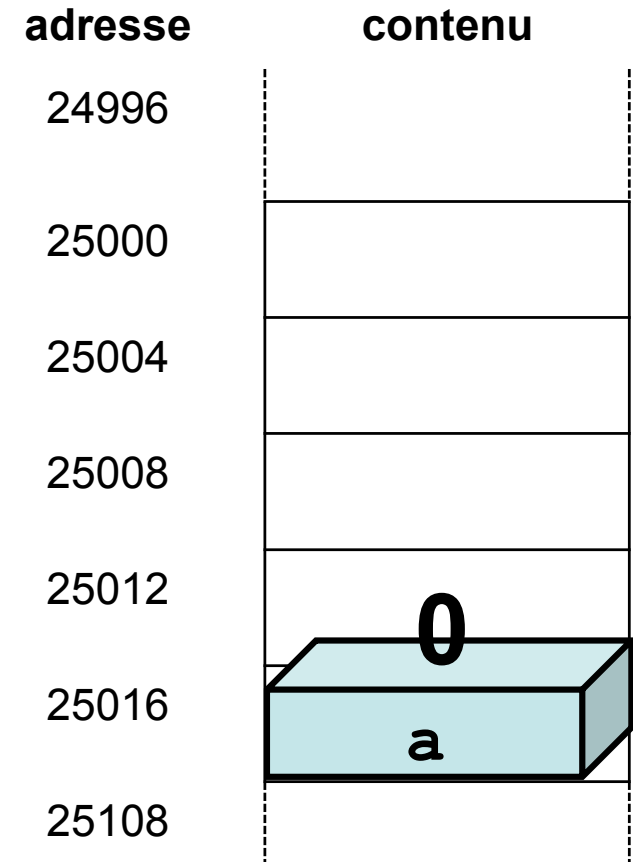
Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    ajoute1(pm);
    ajoute1(pm);
}
```

```
...
int a = 0;
```

→ ajoute2(&a);



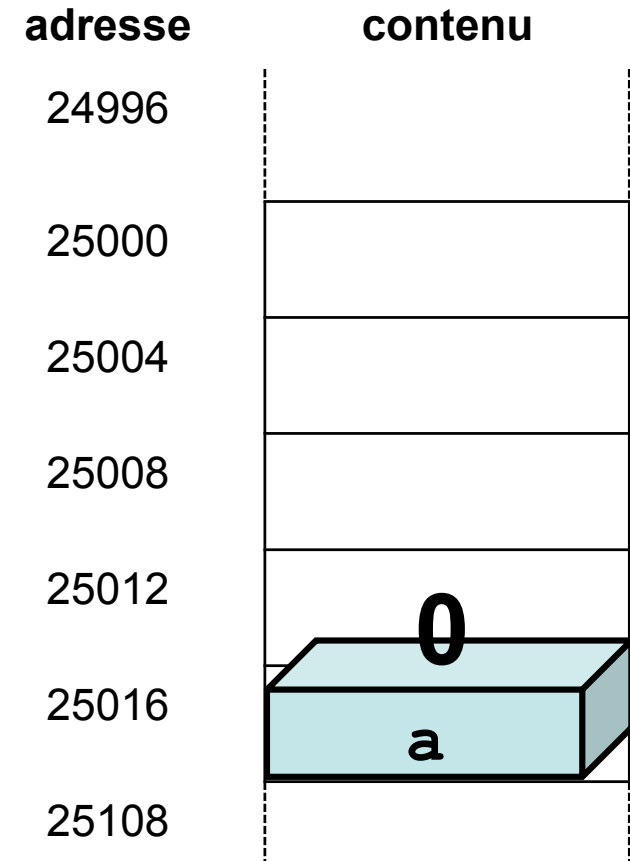
Pas-à-pas

```
void ajoutel1(int * pn)
{
    *pn = *pn + 1;
}
```

```
→ void ajoute2(int * pm)
{
    ajoutel1(pm);
    ajoutel1(pm);
}
```

```
...
int a = 0;
```

```
→ ajoute2(&a);
```



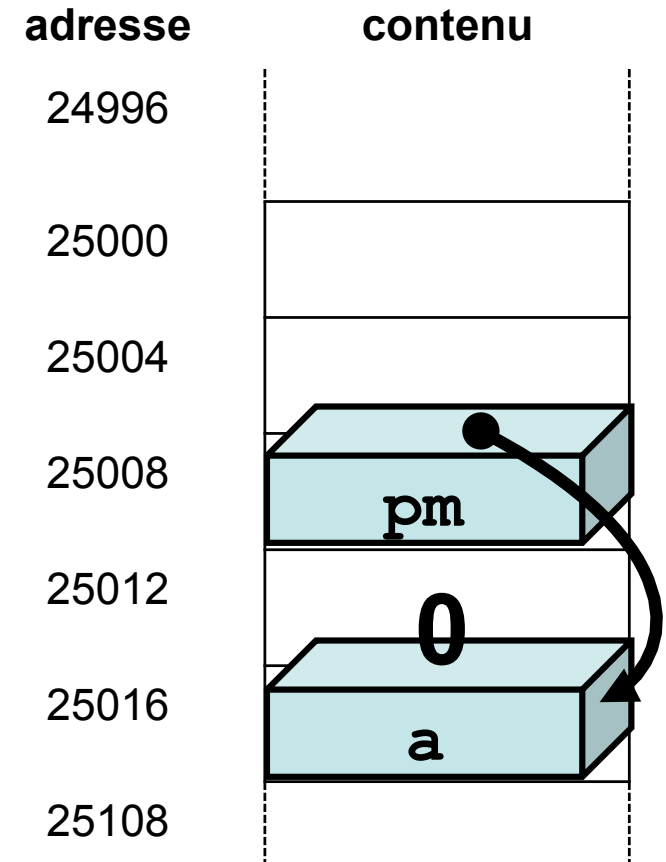
Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

```
→ void ajoute2(int * pm)
{
    ajoute1(pm);
    ajoute1(pm);
}
```

```
...
int a = 0;
```

```
→ ajoute2(&a);
```



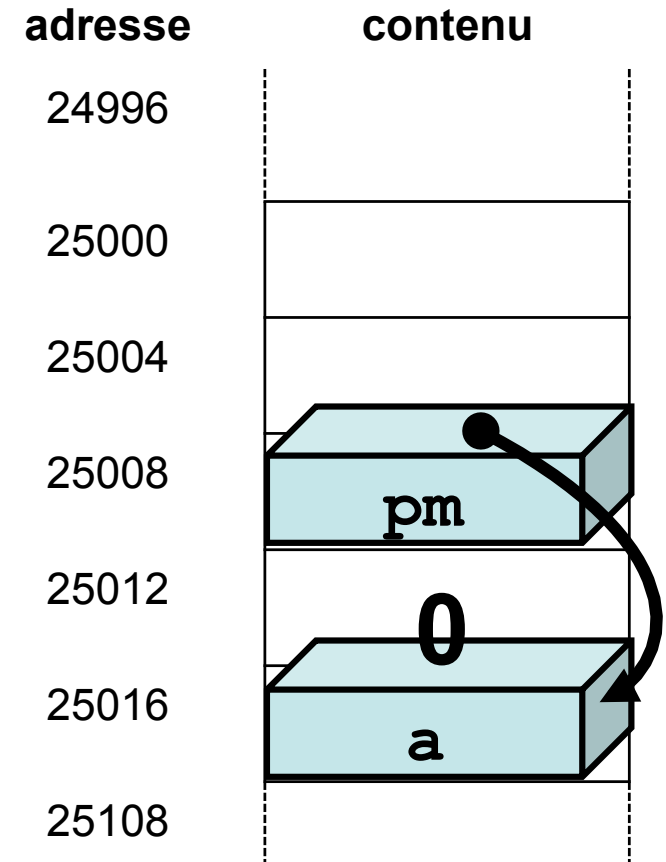
Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    ajoute1(pm);
    ajoute1(pm);
}
```

```
...
int a = 0;
```

```
→ ajoute2(&a);
```

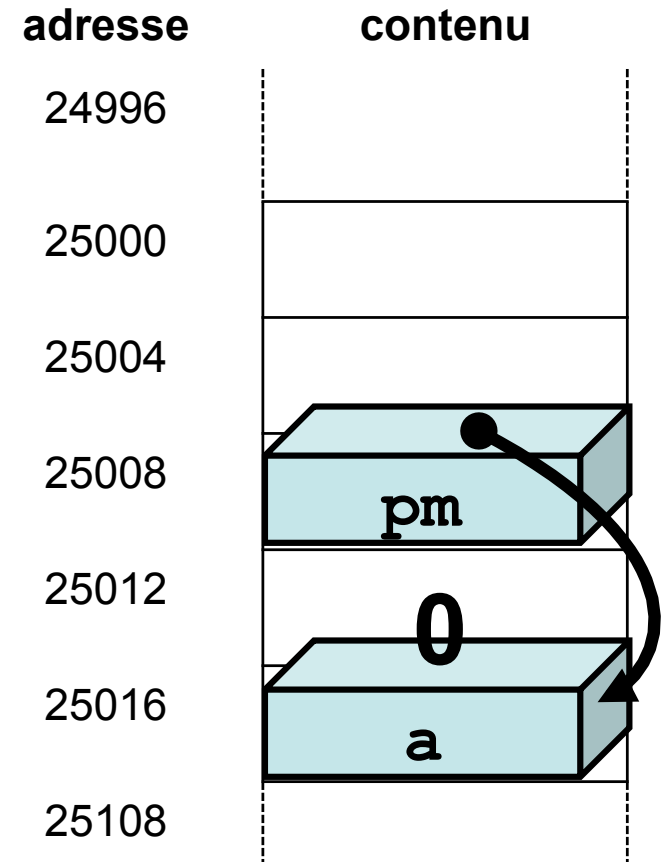


Pas-à-pas

```
→ void ajoutel(int * pn)
  {
    *pn = *pn + 1;
  }

void ajoute2(int * pm)
  {
    → ajoutel(pm);
    ajoutel(pm);
  }

  ...
  int a = 0;
  → ajoute2(&a);
```

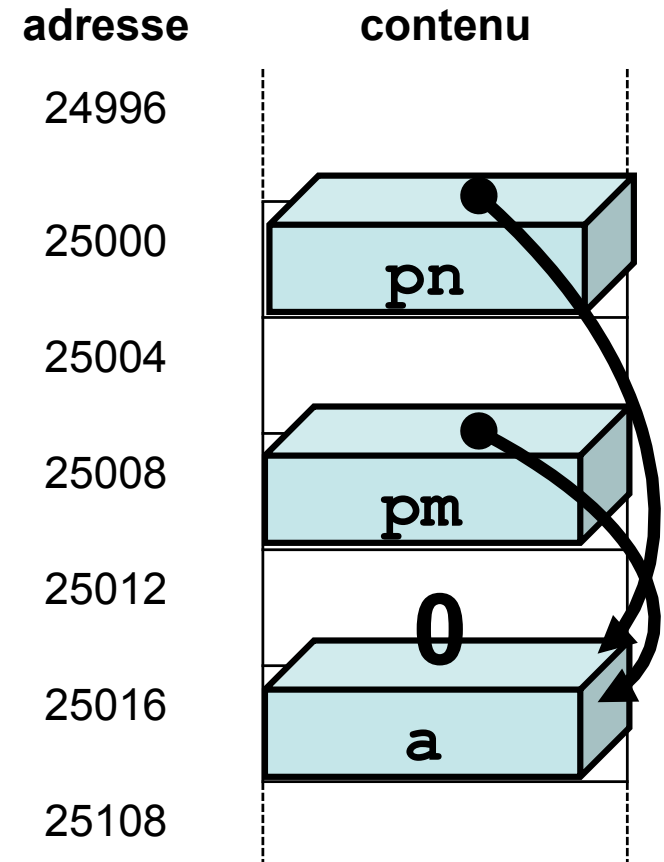


Pas-à-pas

```
→ void ajoutel(int * pn)
  {
    *pn = *pn + 1;
  }

void ajoute2(int * pm)
  {
    → ajoutel(pm);
    ajoutel(pm);
  }

  ...
  int a = 0;
  → ajoute2(&a);
```



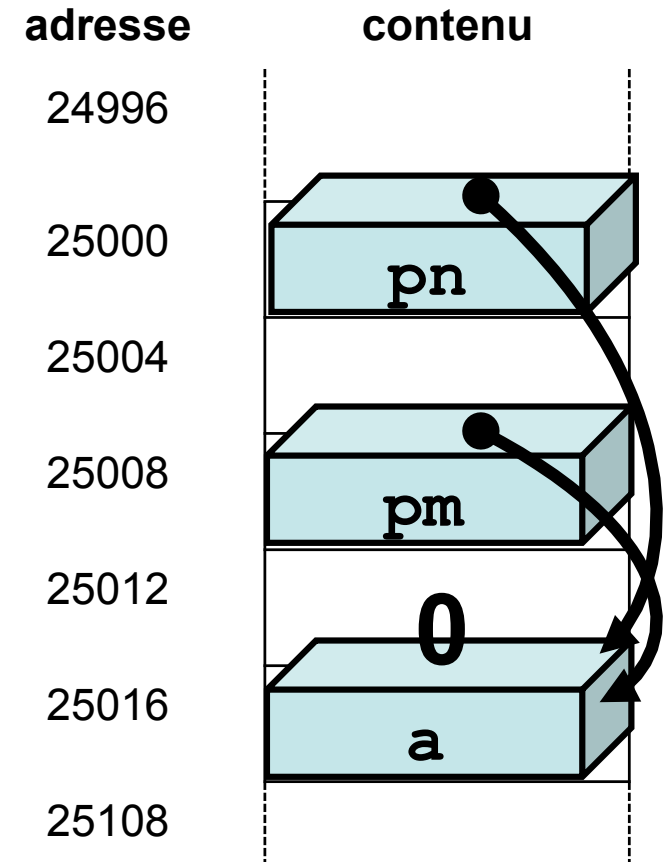
Pas-à-pas

```
void ajoute1(int * pn)
{
  → *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
  → ajoute1(pm);
  ajoute1(pm);
}
```

```
...
int a = 0;
```

```
→ ajoute2(&a);
```



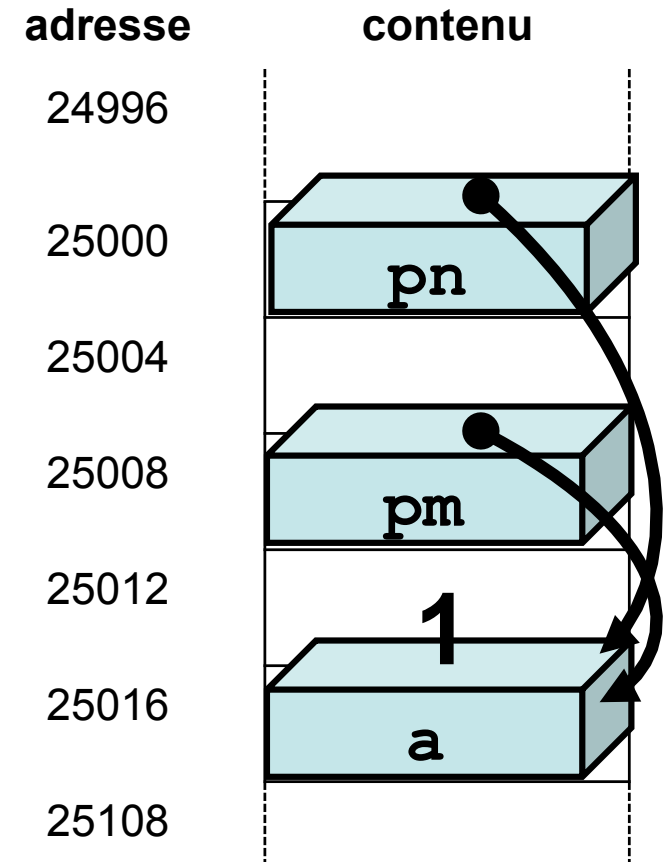
Pas-à-pas

```
void ajoute1(int * pn)
{
  → *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
  → ajoute1(pm);
  ajoute1(pm);
}
```

```
...
int a = 0;
```

```
→ ajoute2(&a);
```



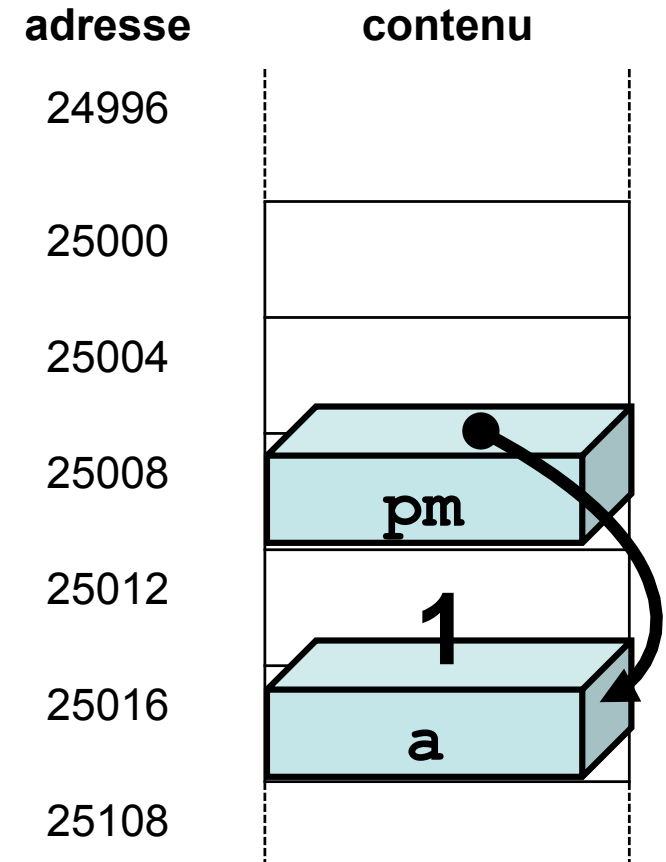
Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    ajoute1(pm);
    ajoute1(pm);
}
```

```
...
int a = 0;
```

```
→ ajoute2(&a);
```



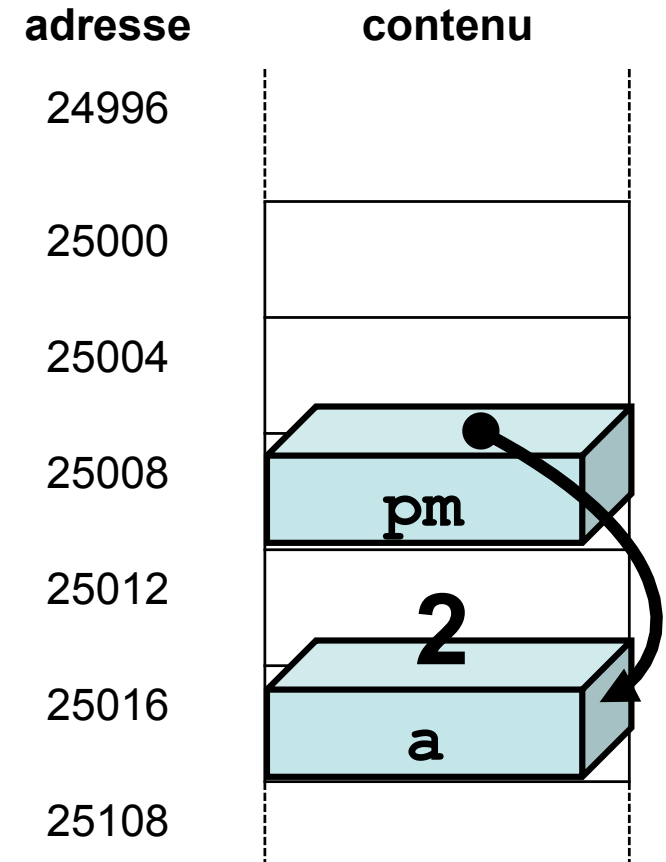
Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

```
void ajoute2(int * pm)
{
    ajoute1(pm);
    ajoute1(pm);
}
```

```
...
int a = 0;
```

```
→ ajoute2(&a);
```

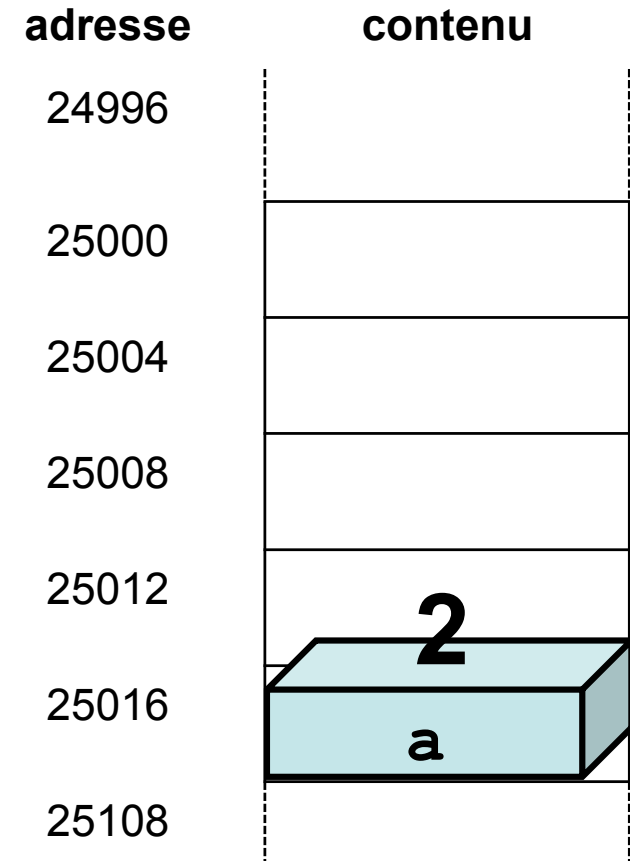


Pas-à-pas

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}

void ajoute2(int * pm)
{
    ajoute1(pm);
    ajoute1(pm);
}

...
int a = 0;
ajoute2(&a);
```



Fonction echange

Pour échanger la valeur de deux variables `a` et `b`, on peut faire:

```
int tmp;  
tmp = a;  
a = b;  
b = tmp;
```

On veut écrire une fonction qui exécute ce code pour inverser deux variables passées en paramètre.

On doit donc utiliser des pointeurs pour ces paramètres.

La fonction a donc comme en-tête:

```
void echange(int * pa, int * pb)
```

Fonction echange

Pour échanger la valeur de deux variables a et b, on peut faire:

```
int tmp;  
tmp = a;  
a = b;  
b = tmp;
```

La fonction echange s'écrit:

```
void echange(int * pa, int * pb)  
{  
    int tmp;  
    tmp = *pa;  
    *pa = *pb;  
    *pb = tmp;  
}
```

et s'appelle:

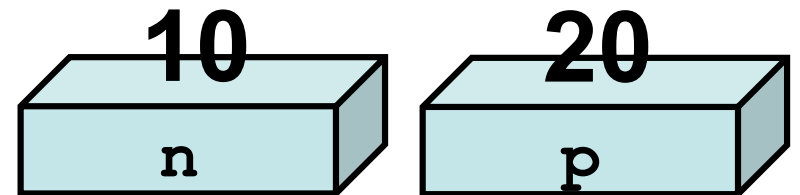
```
int n = 10, p = 20;  
echange(&n, &p);
```

Pas-à-pas

```
void exchange(int * pa, int * pb)
{
    int tmp;
    tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}
```

...

```
→ int n = 10, p = 20;
   exchange(&n, &p);
```

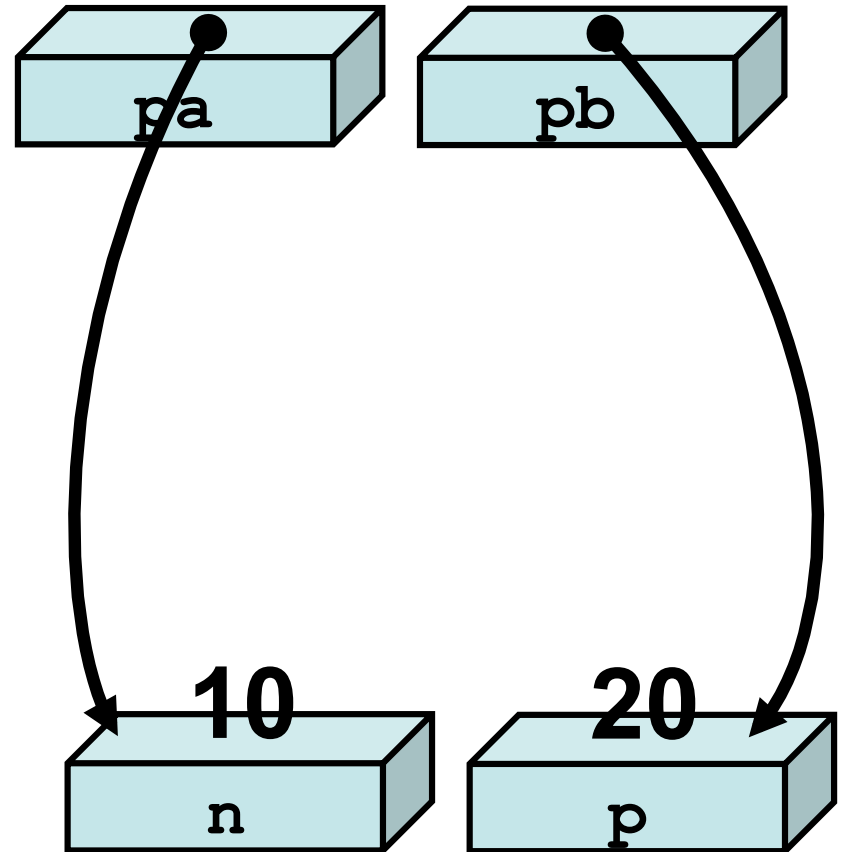


Pas-à-pas

```
→ void exchange(int * pa, int * pb)
{
  int tmp;
  tmp = *pa;
  *pa = *pb;
  *pb = tmp;
}
```

...

```
→ int n = 10, p = 20;
   exchange(&n, &p);
```

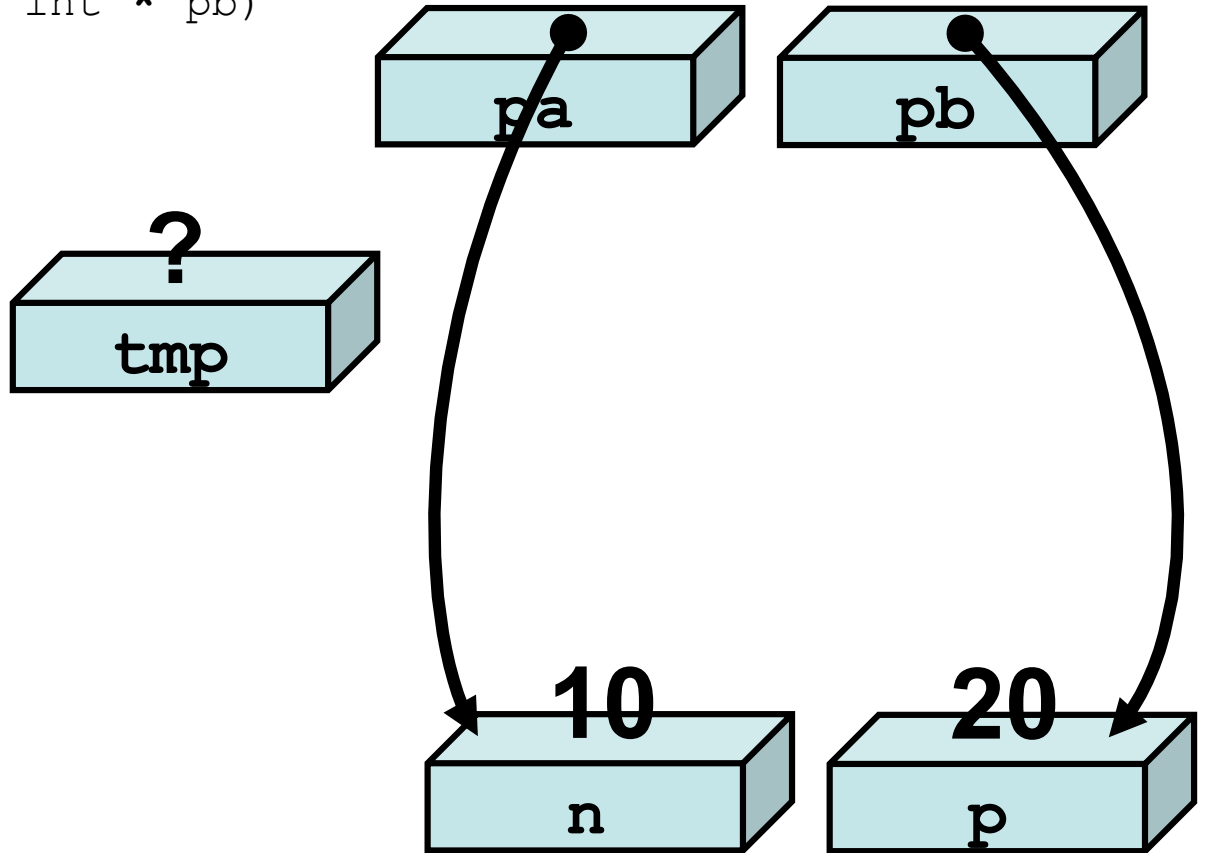


Pas-à-pas

```
void exchange(int * pa, int * pb)
{
  → int tmp;
  tmp = *pa;
  *pa = *pb;
  *pb = tmp;
}
```

...

```
→ int n = 10, p = 20;
   exchange(&n, &p);
```

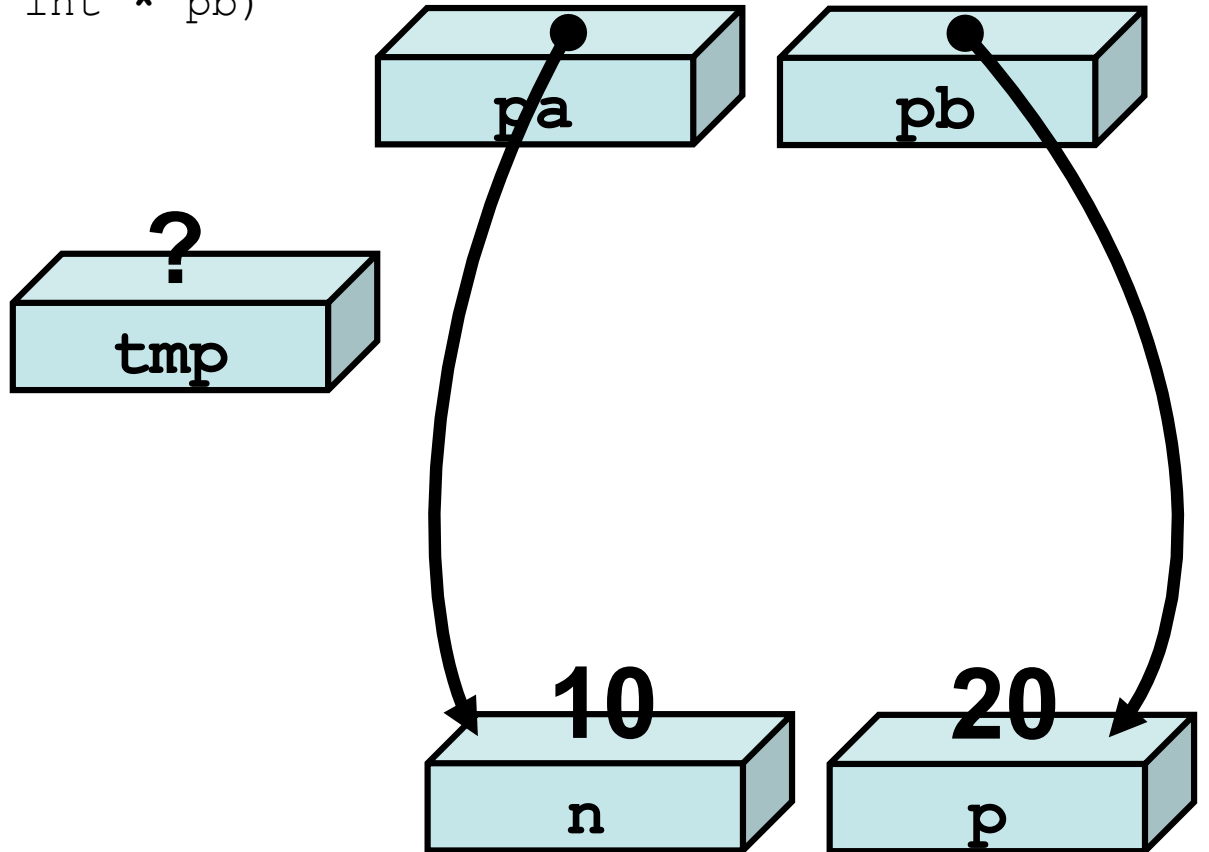


Pas-à-pas

```
void exchange(int * pa, int * pb)
{
    int tmp;
    → tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}
```

...

```
→ int n = 10, p = 20;
   exchange(&n, &p);
```



Pas-à-pas

```
void exchange(int * pa, int * pb)
{
  int tmp;
  tmp = *pa;
  *pa = *pb;
  *pb = tmp;
}
```



```
tmp = *pa;
```

```
*pa = *pb;
```

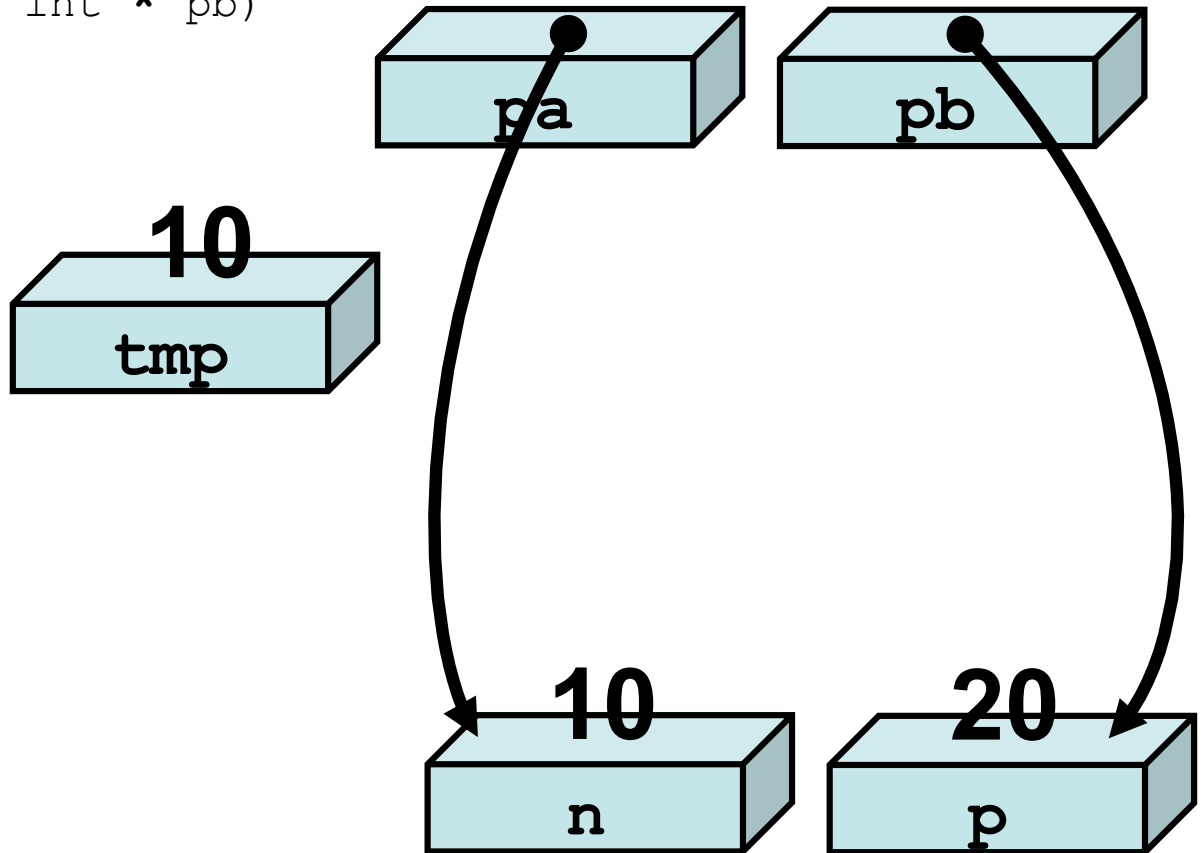
```
*pb = tmp;
```

...



```
int n = 10, p = 20;
```

```
exchange(&n, &p);
```

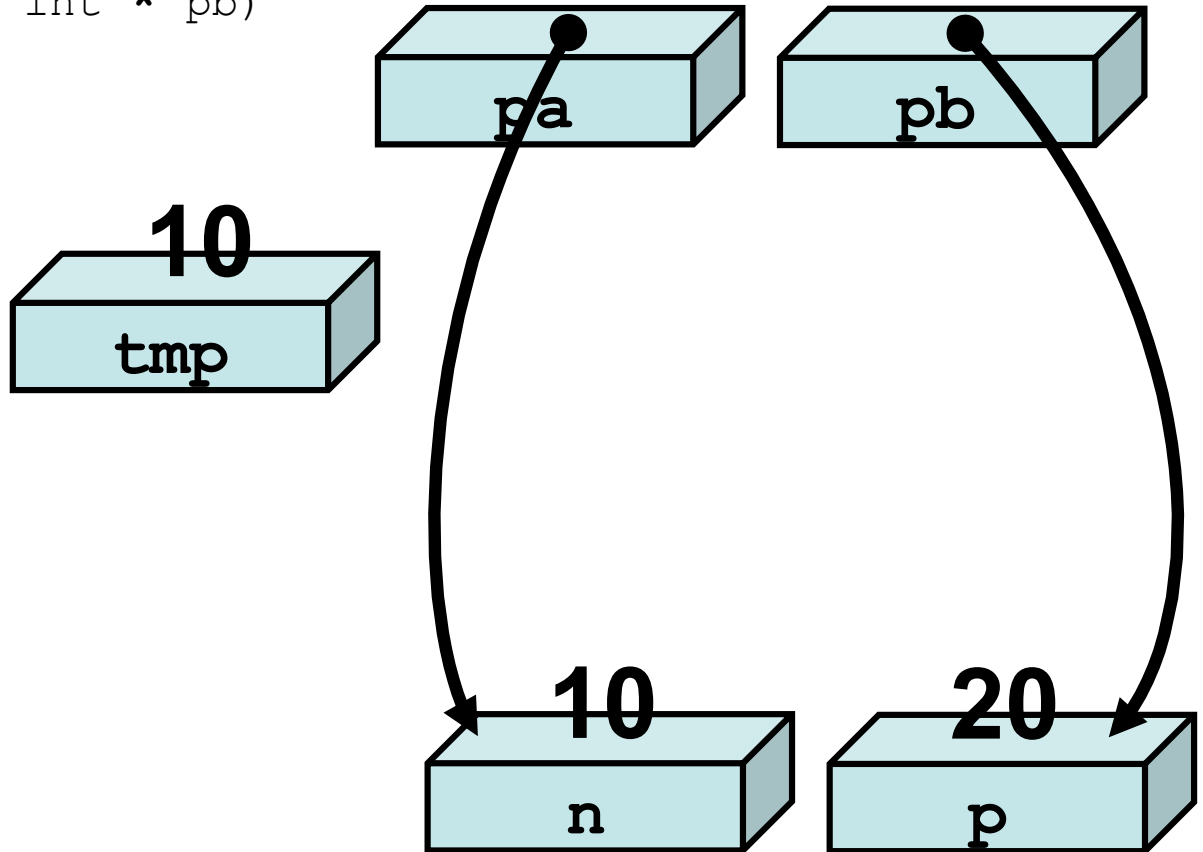


Pas-à-pas

```
void exchange(int * pa, int * pb)
{
    int tmp;
    tmp = *pa;
    → *pa = *pb;
    *pb = tmp;
}
```

...

```
→ int n = 10, p = 20;
   exchange(&n, &p);
```

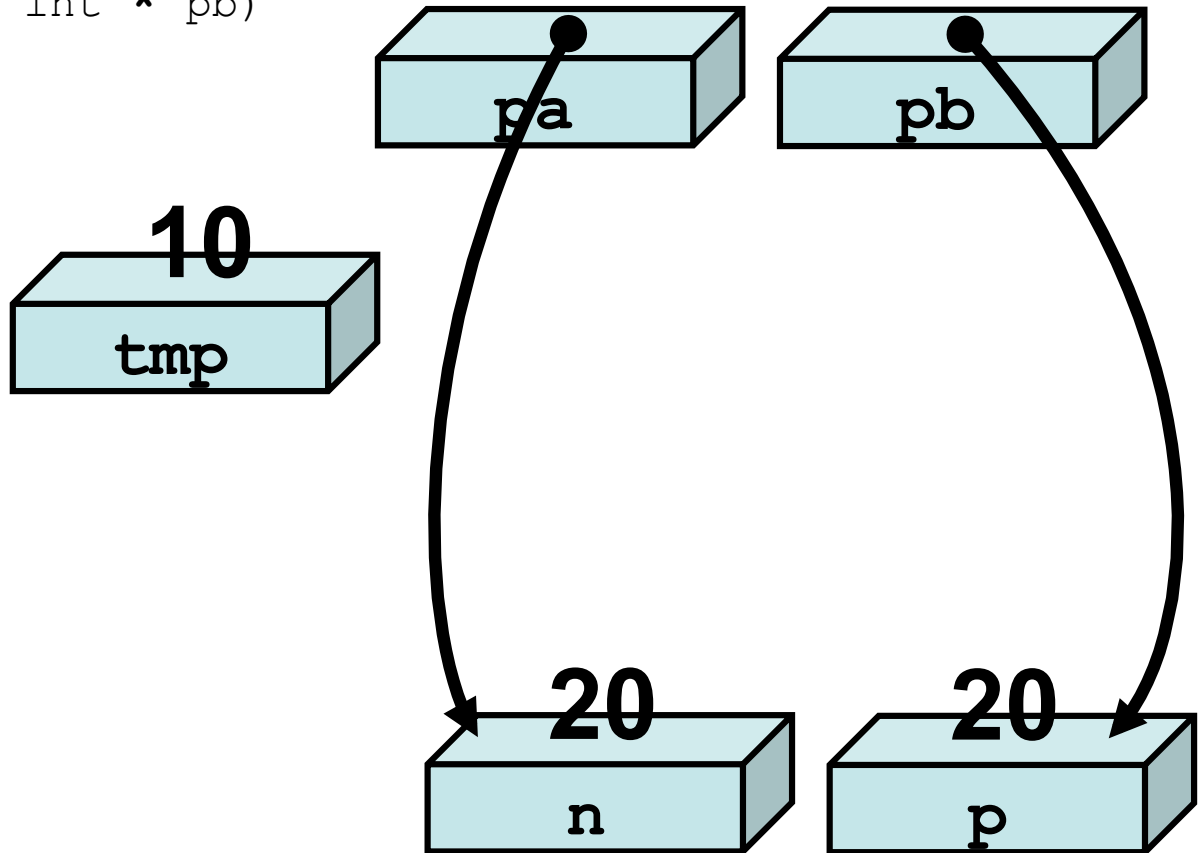


Pas-à-pas

```
void exchange(int * pa, int * pb)
{
    int tmp;
    tmp = *pa;
    → *pa = *pb;
    *pb = tmp;
}
```

...

```
→ int n = 10, p = 20;
   exchange(&n, &p);
```

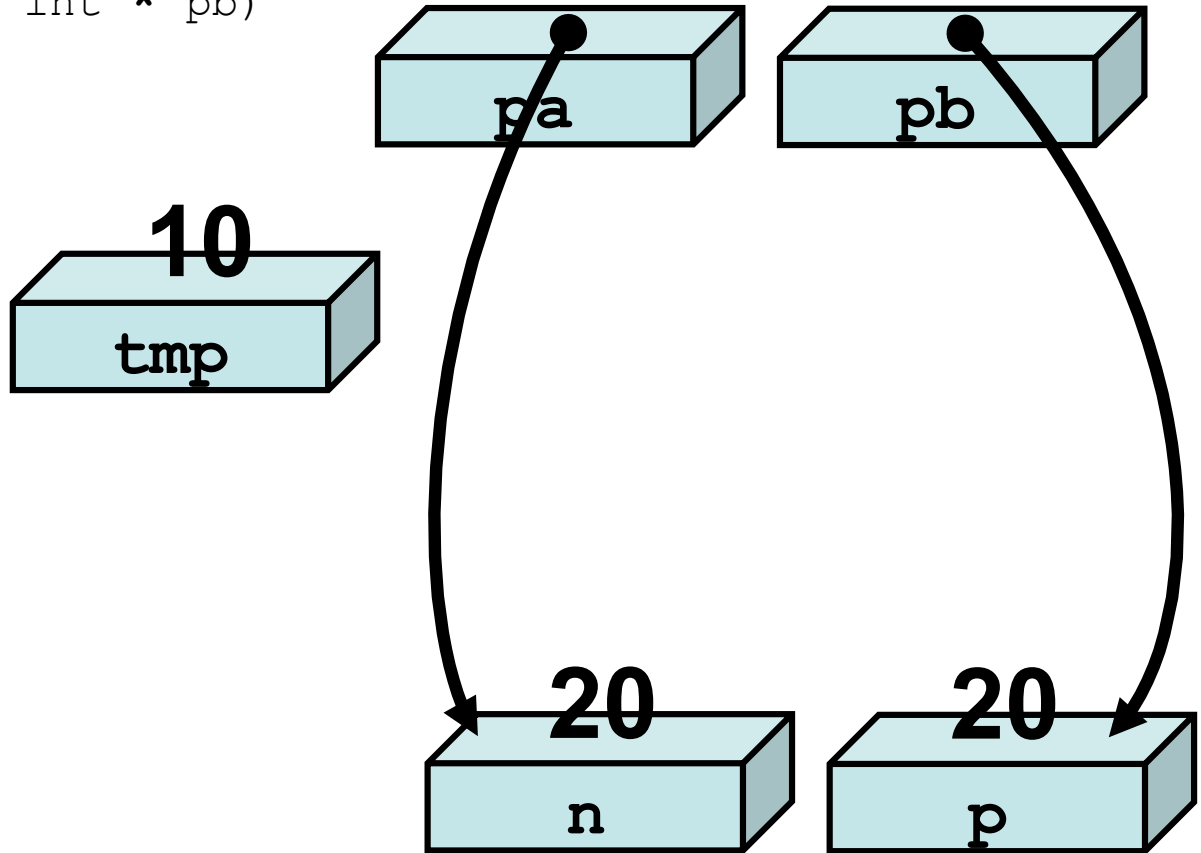


Pas-à-pas

```
void exchange(int * pa, int * pb)
{
    int tmp;
    tmp = *pa;
    *pa = *pb;
    → *pb = tmp;
}
```

...

```
→ int n = 10, p = 20;
   exchange(&n, &p);
```

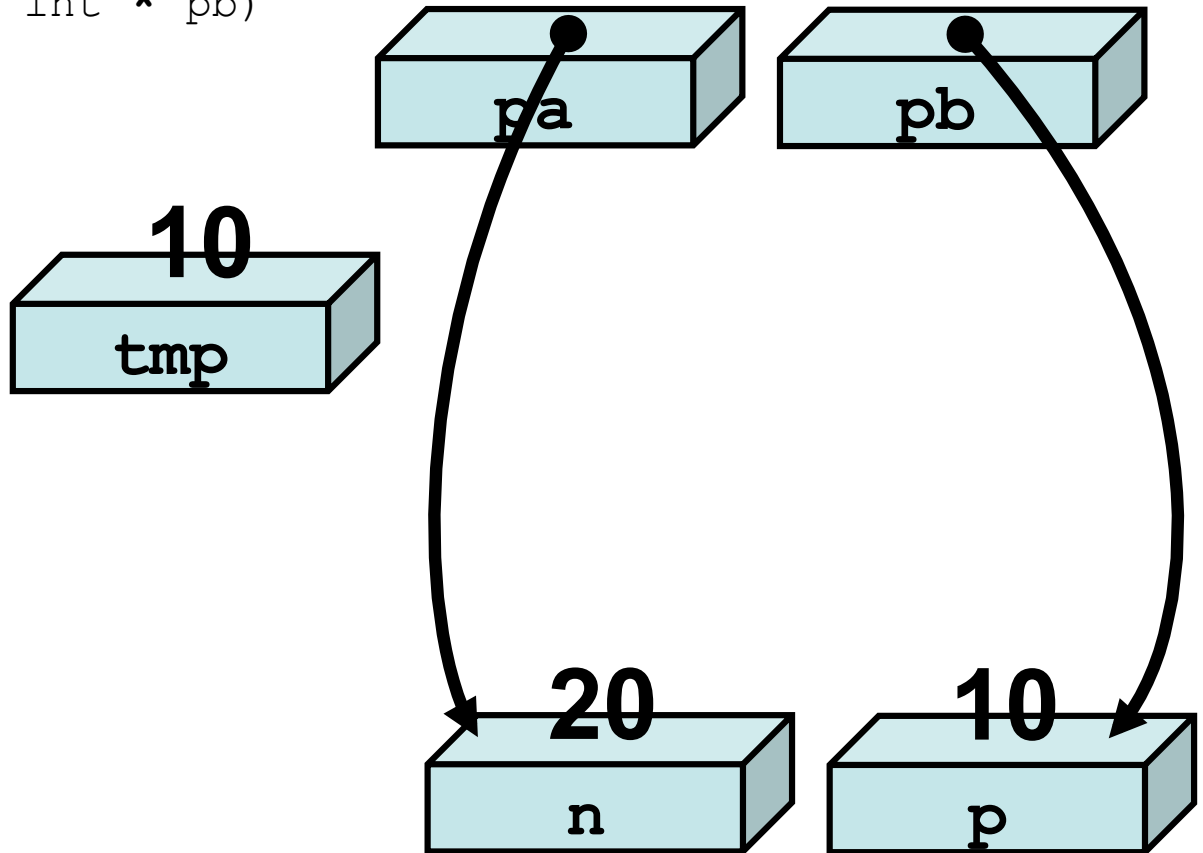


Pas-à-pas

```
void exchange(int * pa, int * pb)
{
    int tmp;
    tmp = *pa;
    *pa = *pb;
    → *pb = tmp;
}
```

...

```
→ int n = 10, p = 20;
   exchange(&n, &p);
```

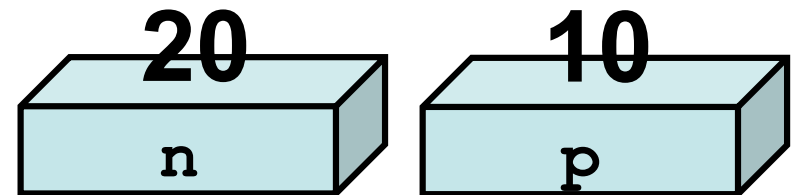


Pas-à-pas

```
void exchange(int * pa, int * pb)
{
    int tmp;
    tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}
```

...

```
int n = 10, p = 20;
exchange(&n, &p);
```



Exercise

```
void f(int * p, int * q)
{
    int * r = p;
    p = q;
    q = r;
}

...
int a = 1, b = 2;
f(&a, &b);
cout << a << " " << b << endl;
```

Deuxième application
Une fonction peut accéder
aux éléments d'un tableau
passé en paramètre

Tableaux et pointeurs

Utilisé seul, le nom du tableau correspond à l'adresse du premier élément.

Par exemple, on peut faire:

```
int T[5];  
int * tab = T;
```

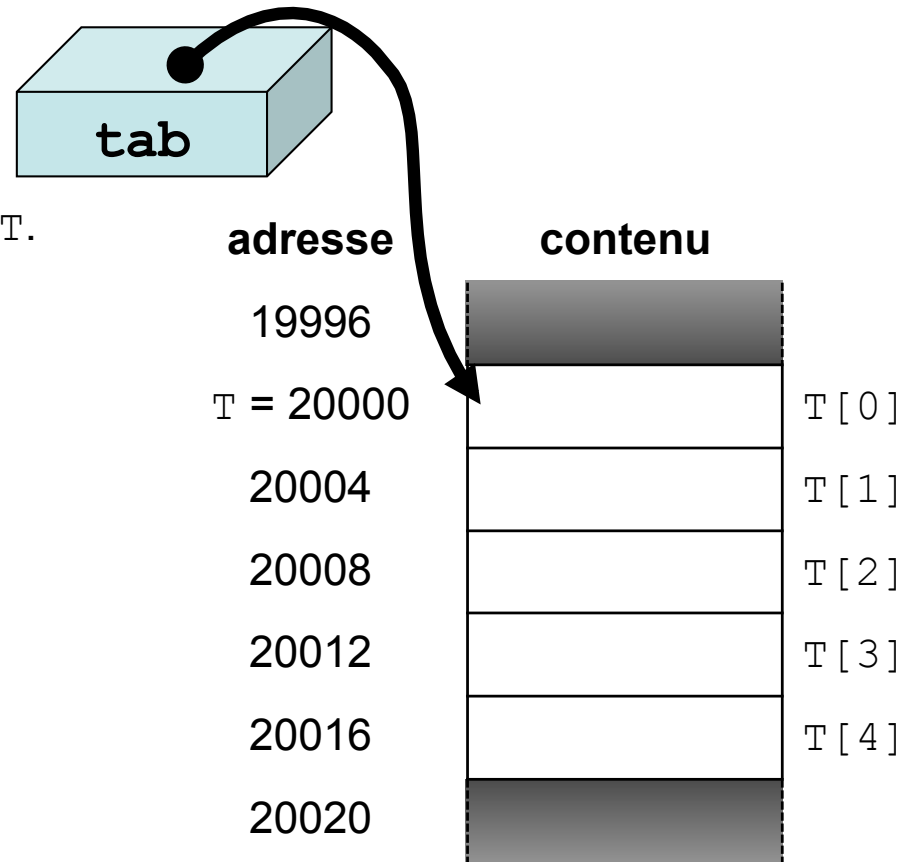
et `tab` pointe alors sur le premier élément de `T`.

Attention, `T` n'est pas pour autant un pointeur.

On ne peut pas faire par exemple:

```
T = &a;
```

si `T` a été déclaré comme un tableau.



Tableaux et pointeurs

Après:

```
int T[5];  
int * tab = T;
```

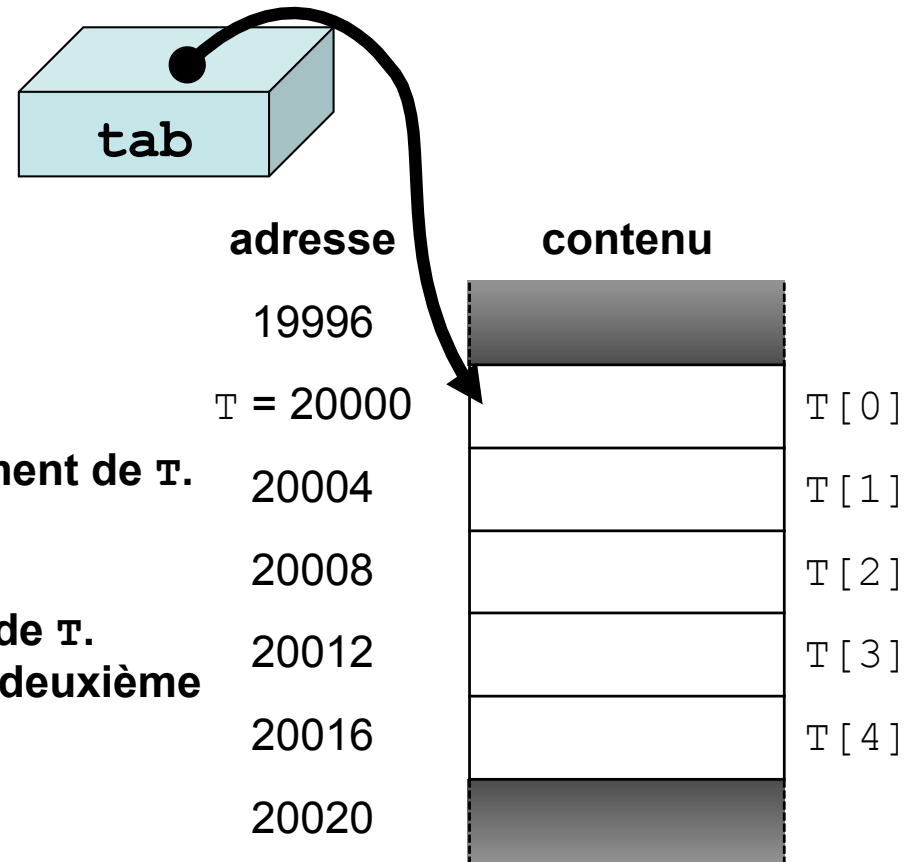
`tab` pointe sur le premier élément de `T`.

***`tab` correspond donc à `T[0]`, le premier élément de `T`.**

`tab + 1` est l'adresse du deuxième élément de `T`.

***(`tab + 1`) correspond donc à `T[1]`, le deuxième élément de `T`.**

etc...



Notez que:

Si

`tab` vaut 20000,
et si `tab` est un pointeur sur `int`

Alors

`tab + 1` vaut 20004

pour pointer sur le `int` suivant: un `int` est représenté sur 4 octets, et les adresses sont définies en octet.

Ce mécanisme de calcul simplifie l'accès aux éléments du tableau: il n'y a pas à savoir qu'il faut multiplier par 4 dans le cas des `int`, par 8 dans le cas des `double`, etc...

Tableaux et pointeurs

Après:

```
int T[5];  
int * tab = T;
```

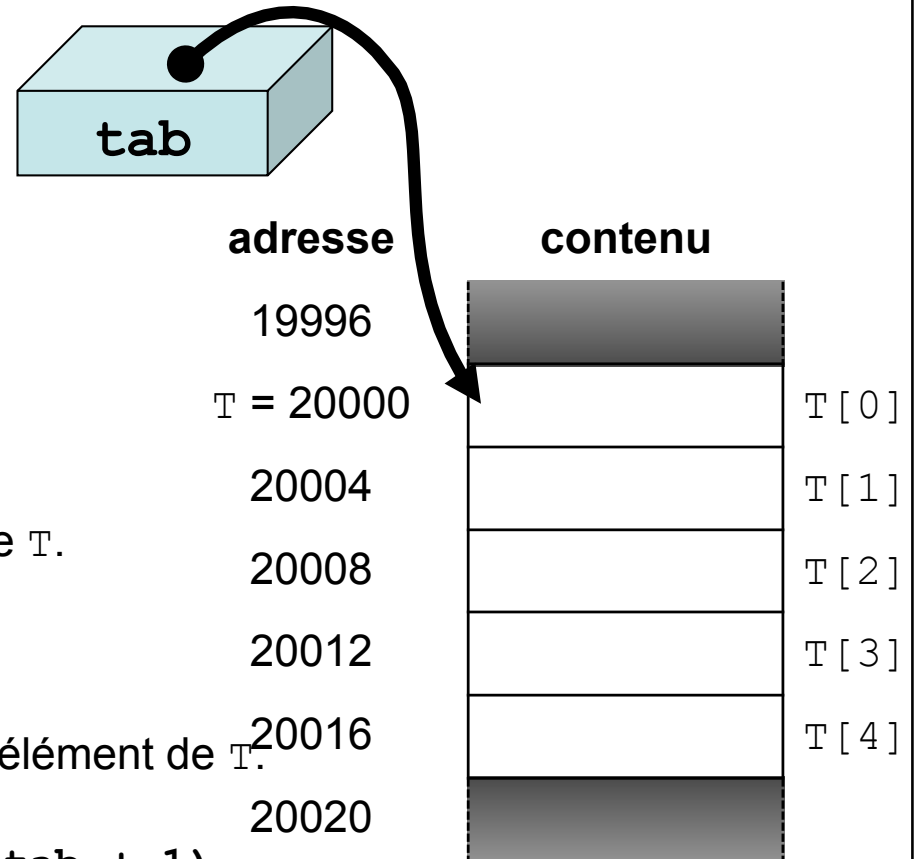
`tab` pointe sur le premier élément de `T`.

`*tab` correspond donc à `T[0]`, le premier élément de `T`.

`tab + 1` est l'adresse du deuxième élément de `T`.

`*(tab + 1)` correspond donc à `T[1]`, le deuxième élément de `T`.

On peut également écrire `tab[1]` à la place de `*(tab + 1)`.



En C et en C++, on peut écrire

```
tab[i]
```

à la place de

```
*(tab + i)
```

ce qui permet d'utiliser `tab` comme on utiliserait un tableau!

Accès aux éléments d'un tableau

Supposons qu'on veut écrire une fonction qui ajoute 1 aux éléments d'un tableau. On peut donc écrire cette fonction comme ça:

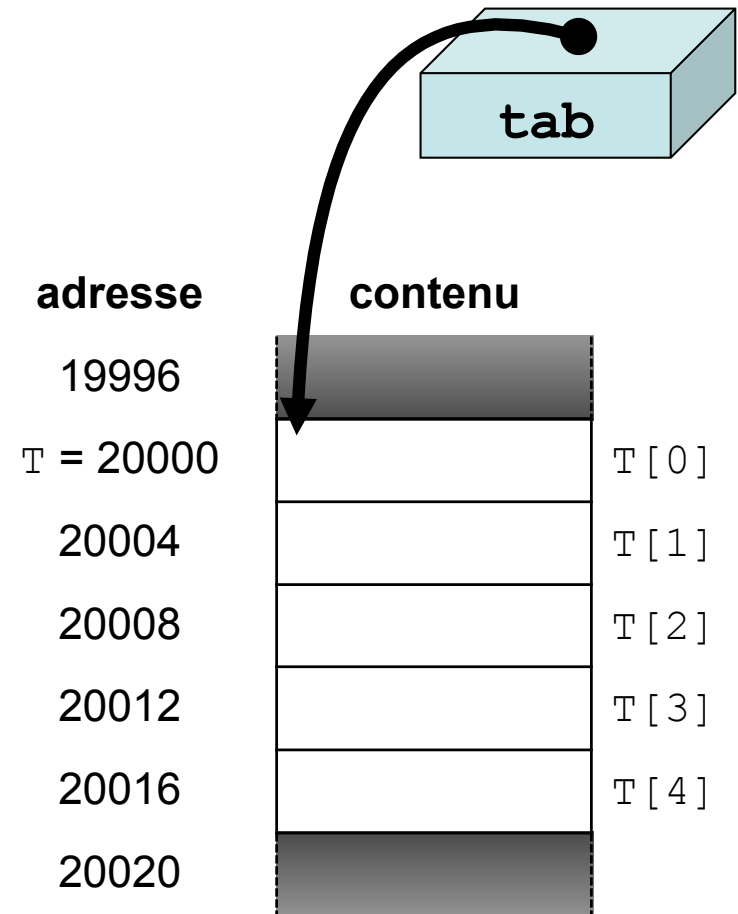
```
void ajoutel_aux_elements(int * tab)
{
    for(int i = 0; i < 5; i++)
        *(tab + i) = *(tab + i) + 1;
}
```

ou comme ça:

```
void ajoutel_aux_elements(int * tab)
{
    for(int i = 0; i < 5; i++)
        tab[i] = tab[i] + 1;
}
```

et on peut appeler cette fonction ainsi:

```
int T[5];
ajoutel_aux_elements(T);
```



On peut écrire indifféremment:

```
void ajoute1_aux_elements(int tab[5])
```

ou

```
void ajoute1_aux_elements(int * tab)
```

Le 5 du premier en-tête N'est PAS considéré par le compilateur.

On peut même écrire:

```
void ajoute1_aux_elements(int tab[])
```

- `void ajoutel_aux_elements(int tab[5])`
- `void ajoutel_aux_elements(int tab[])`
- `void ajoutel_aux_elements(int * tab)`

→ Ces trois façons sont gérées *exactement* de la même façon par le compilateur:

Dans les 3 cas, `tab` est considéré comme un pointeur sur `int`.

A l'appel de la fonction, par exemple:

```
ajoutel_aux_elements(T);
```

`tab` pointe sur le premier élément de `T`.

Parce que `tab[i]` est la même chose que `*(tab + i)`, on peut utiliser `tab` comme un tableau, même s'il est défini comme un pointeur dans la déclaration:

```
void ajoutel_aux_elements(int * tab)
```

Tableau ou pointeur en paramètre

DORENAVANT, UTILISEZ LA NOTATION:

```
void ajoute1_aux_elements(int * tab)
```

pour définir un paramètre correspondant à un tableau.

MAIS ATTENTION:

CECI EST VRAI UNIQUEMENT POUR LES TABLEAUX A UNE DIMENSION.

CE N'EST PAS VRAI POUR LES TABLEAUX A DEUX DIMENSIONS (OU PLUS)
QUE NOUS VERRONS DANS LA SUITE DE CE COURS.

Attention au nombre d'éléments

La fonction

```
void ajoute1_aux_elements(int * tab)
```

ne connaît PAS LA TAILLE du tableau passé en paramètre.

Passer le nombre d'éléments en paramètre

Pour que la fonction puisse être utilisée pour n'importe quel tableau, quelle que soit sa taille, on peut passer la taille du tableau en paramètre.

Exemple:

```
void ajoute1_aux_elements(int * tab, int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
        tab[i] = tab[i] + 1;
}
```

Exemple d'utilisation:

```
int T[5], U[101];
```

...

```
ajoute1_aux_elements(T, 5);
ajoute1_aux_elements(U, 101);
```

Passer l'adresse d'un élément d'un tableau en paramètre

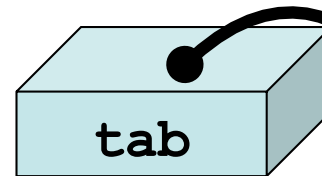
Supposons maintenant qu'on veuille écrire la fonction `ajoute1_aux_elements()` en utilisant la fonction `ajoute1()` vue précédemment:

```
void ajoute1(int * pn)
{
    *pn = *pn + 1;
}
```

On peut écrire:

```
void ajoute1_aux_elements(int * tab, int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
        ajoute1(&(tab[i]));
}
```

Passer l'adresse d'un élément d'un tableau en paramètre



`&(tab[i])`

correspond à l'adresse de l'élément i de `tab`.

Une autre façon d'obtenir cette adresse est:

`tab + i`

`&(tab[i])` est équivalent à `tab + i`

adresse	contenu
19996	
$T = 20000$	T[0]
20004	T[1]
20008	T[2]
20012	T[3]
20016	T[4]
20020	

Passer l'adresse d'un élément d'un tableau en paramètre

```
void ajoutel_aux_elements(int * tab, int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
        ajoutel(tab + i);
}
```

est donc équivalent à (mais plus élégant que):

```
void ajoutel_aux_elements(int * tab, int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
        ajoutel(&(tab[i]));
}
```

Quelques exemples

```
void fonction1(int * tab, int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
        cout << tab[i] << endl;
}
```

```
int fonction2(int * tab, int nb_elements)
{
    int f = 0;

    for(int i = 0; i < nb_elements; i++)
        f = f + tab[i];

    return f;
}
```

```
void fonction3(float * tab, int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
        tab[i] = 0.0;
}
```

```
void fonction4(float * X, float * Y, int nb_elements)
{
    for(int i = 0; i < nb_elements; i++)
        Y[i] = cos(X[i]);
}
```

Exercise

```
void f(int * q, int r)
{
    q = q + r;
    *q = 0;
}

void g(int * q, int r)
{
    q[r] = 0;
}

...

int T[4] = {1, 5, 6, 9};
int * p = T;

*p = *p + 1;
cout << T[0] << " " << T[1] << endl;

p = p + 1;
cout << T[0] << " " << T[1] << endl;

*p = *p + 1;
cout << T[0] << " " << T[1] << endl;

f(T, 2);
cout << T[2] << " " << T[3] << endl;

g(T, 3);
cout << T[2] << " " << T[3] << endl;
```

Troisième application

Allocation dynamique:
Allouer de la mémoire
pendant l'exécution du programme

Allouer de la mémoire pendant l'exécution du programme

On peut allouer de la mémoire pendant l'exécution du programme.
On parle d'**Allocation dynamique**.

L'allocation se fait avec l'instruction `new`.

Une fois que la mémoire allouée n'est plus utile, IL FAUT la libérer, avec l'instruction `delete`.

Allouer de la mémoire pendant l'exécution du programme

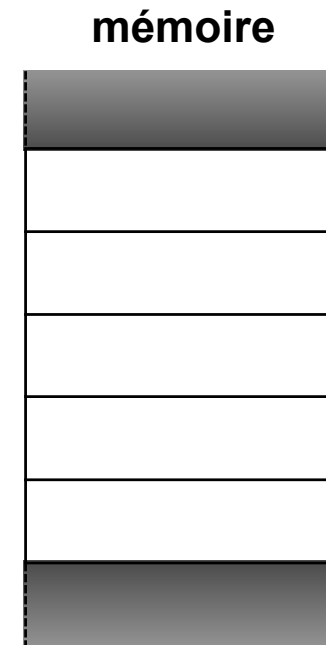
```
int * tab;
```

```
tab = new int[5];
```



L'instruction `new` fait 2 choses:

1. Elle alloue de la mémoire, c'est-à-dire qu'elle réserve une partie de la mémoire de l'ordinateur pour que le programme puisse l'utiliser.



Allouer de la mémoire pendant l'exécution du programme

```
int * tab;
```

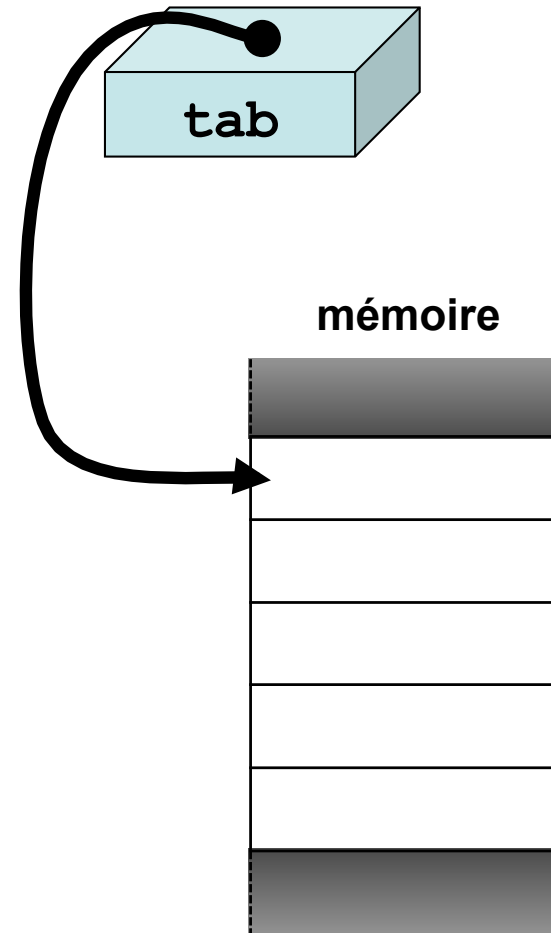
```
tab = new int[5];
```

L'instruction `new` fait 2 choses:

1. Elle alloue de la mémoire, c'est-à-dire qu'elle réserve une partie de la mémoire de l'ordinateur pour que le programme puisse l'utiliser.

2. Elle renvoie l'adresse du début de cette mémoire.

Dans notre exemple, `tab` va donc pointer sur cette mémoire, et permettre au programme de l'utiliser.

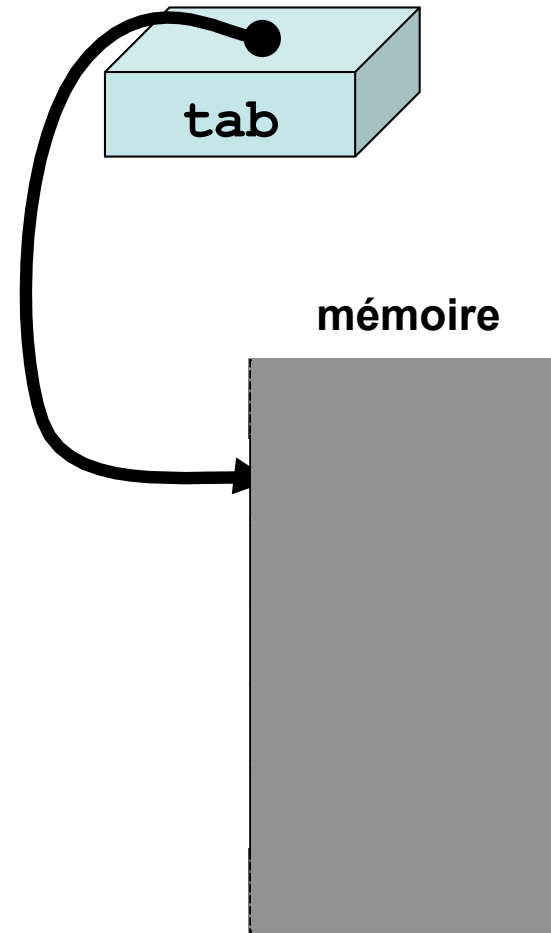


Allouer de la mémoire pendant l'exécution du programme

```
int * tab;  
  
tab = new int[5];  
  
tab[2] = 10;  
tab[3] = 5 * tab[2];  
  
delete [] tab;
```

L'instruction `delete` libère la mémoire allouée par `new`, c'est-à-dire que cette partie de la mémoire va pouvoir être utilisée par d'autres programmes.

Le programme n'a plus le droit d'accéder à cette partie de la mémoire.



new **et** delete

Allocation de la mémoire:

```
tab = new int [5];
```

Libération de la mémoire:

```
delete [] tab;
```

new et delete

Allocation de la mémoire:

Nombre d'éléments à allouer

```
tab = new int [5];
```

Retourne l'adresse du premier élément alloué: `T` doit être déclaré comme un pointeur sur `int`.

Type des éléments

Libération de la mémoire:

Adresse de début de la mémoire à libérer

```
delete [] tab;
```

Attention à ne pas oublier les crochets []

Exemple

```
void set(int * tab, int nb_elements, int v)
{
    for(int i = 0; i < nb_elements; i++)
        tab[i] = v;
}
```

```
int main(int argc, char ** argv)
{
    int * T = new int[10];

    // On met tous les elements de T a 0
    set(T, 10, 0);

    //..

    // il ne faut pas oublier de liberer la memoire allouee pour T
    delete [] T;

    return 0;
}
```

Fonction qui alloue un tableau

```
int * cree_tableau(int nb_elements)
{
    int * T;
    T = new int[nb_elements];
    return T;
}
```

Le résultat de la fonction est un pointeur sur le premier élément du tableau.

Exemple d'utilisation:

```
int * T = cree_tableau(10);
```

Fonction qui alloue un tableau

le résultat de la fonction est
de type pointeur sur `int`

```
int * cree_tableau(int nb_elements)
{
    int * T;
    T = new int[nb_elements];
    return T;
}
```

la fonction renvoie l'adresse du
premier élément du tableau



Le résultat de la fonction est un pointeur sur le premier
élément du tableau.

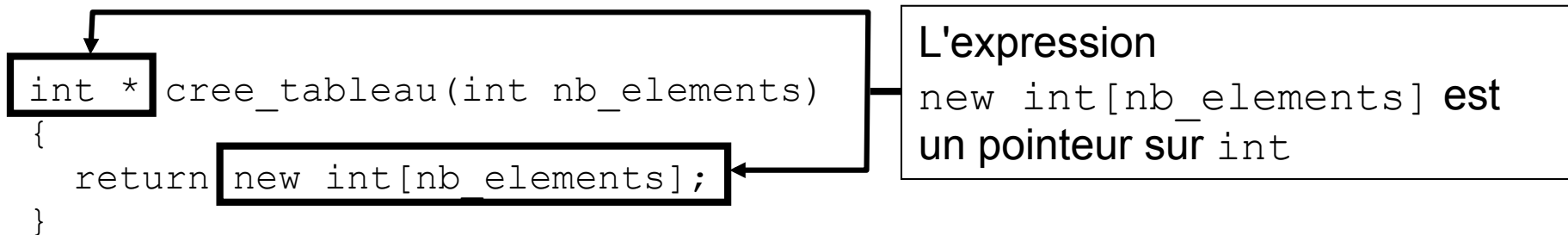
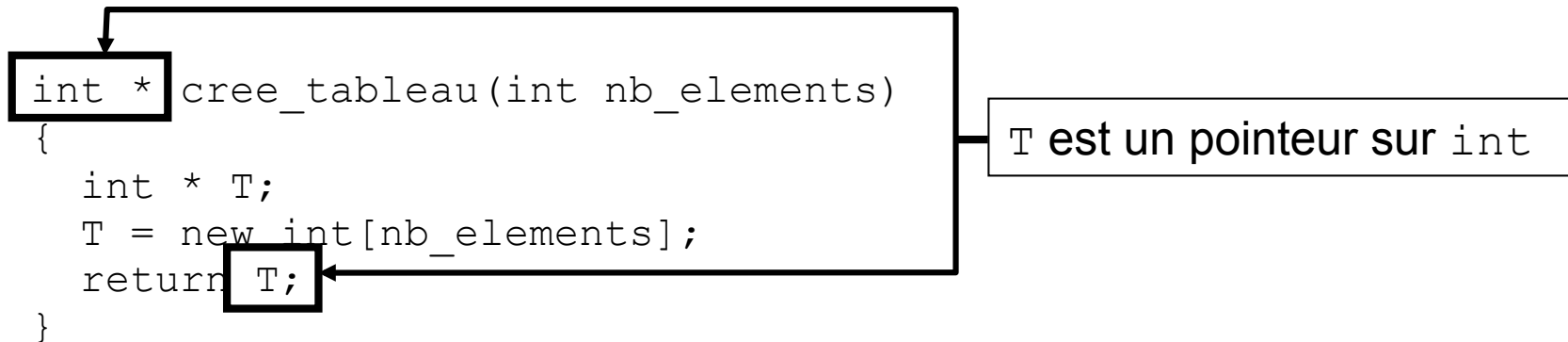
Exemple d'utilisation:

```
int * T = cree_tableau(10);
```

On peut aussi écrire `cree_tableau` de façon plus directe:

```
int * cree_tableau(int nb_elements)
{
    return new int[nb_elements];
}
```

Le type de la fonction est cohérent avec le type de son résultat



Autre fonction qui alloue un tableau

Fonction qui alloue et remplit un tableau avec des valeurs aléatoires:

```
int * cree_tableau_aleatoire(int nb_elements, int val_max)
{
    int * resultat = new int[nb_elements];

    for(int i = 0; i < nb_elements; i++)
        resultat[i] = rand() % val_max;

    return resultat;
}
```

Exemple d'utilisation:

```
int * T = cree_tableau_aleatoire(10, 100);
```

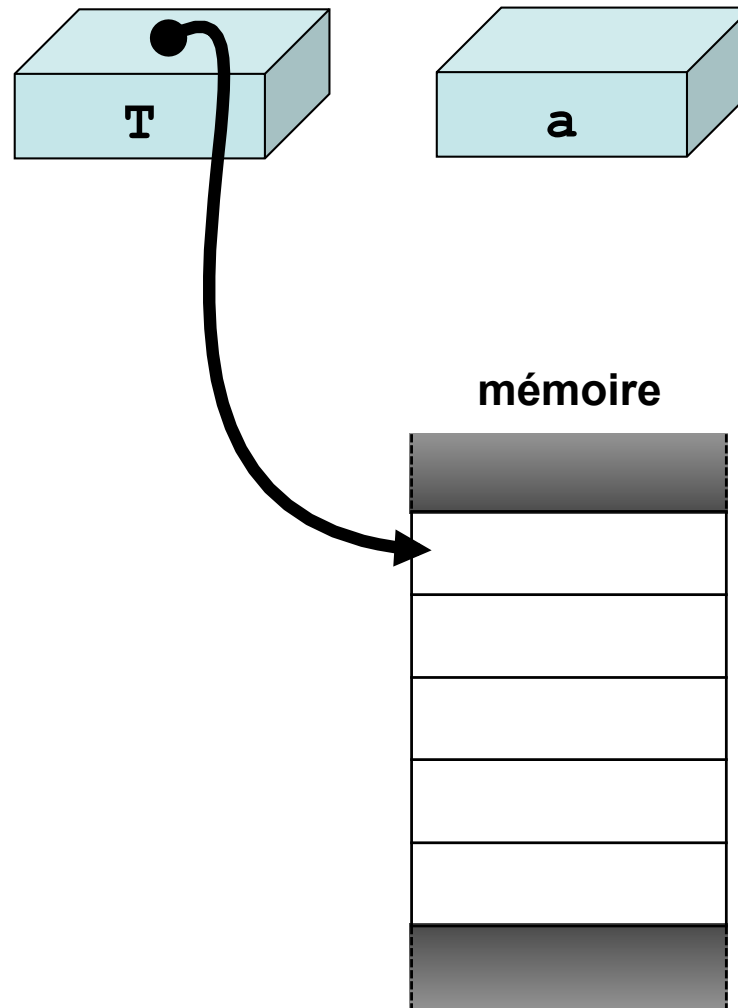
Attention aux *memory leaks* !

Exemple: on commence par faire:

```
int * T;
```

```
int a;
```

```
T = new int [5];
```



Attention aux *memory leaks* !

Exemple:

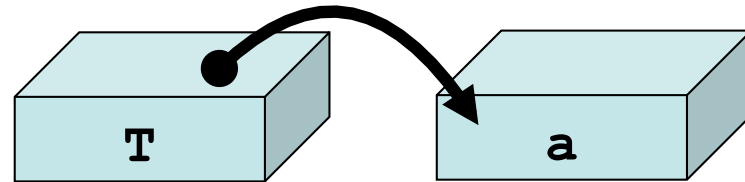
```
int * T;  
int a;
```

```
T = new int [5];
```

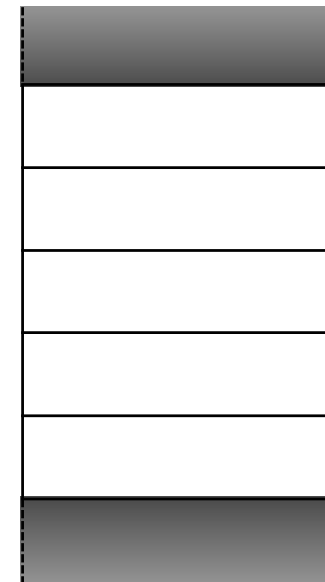
Puis on fait pointer T sur a:

```
T = &a;
```

On ne peut plus libérer la mémoire allouée pour T,
puisque l'on ne sait plus où elle commence !



mémoire



Exercise

```
int * f(int n)
{
    int * p = new int[n];
    for(int i = 0; i < n; i++)
        p[i] = i;
    return p;
}
```

```
int * g(int n)
{
    int p[3];
    for(int i = 0; i < n; i++)
        p[i] = i;
    return p;
}
```

...

```
int * q = f(3);
cout << q[0] << " " << q[1] << " " << q[2] << endl;
```

```
int * r = g(3);
cout << r[0] << " " << r[1] << " " << r[2] << endl;
```