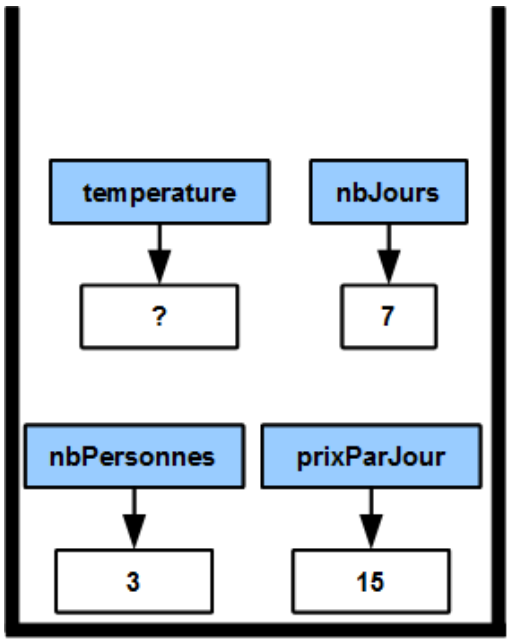
Avant d'aborder ce qui est au cœur de ce chapitre, les tableaux, il est nécessaire en C++ de parler un peu plus de l'organisation de la mémoire. Il est important d'essayer de comprendre ce qui est expliqué dans ce cours mais vous n'avez pas besoin de tout retenir, les idées générales suffiront. Vous pourrez revenir lire ce cours plusieurs fois.

Bits, octets, mémoire et variables

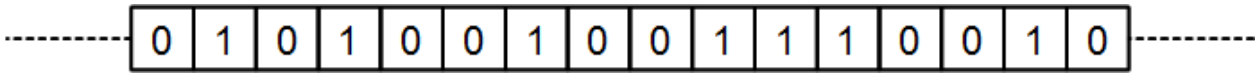
Nous avions déjà expliqué comment, pour l'extrait de programme suivant

```
int temperature;
int nbJours = 7;
int nbPersonnes = 3;
int prixParJour = 15;
```

on pouvait représenter l'état des variables à l'aide d'un dessin de ce type :



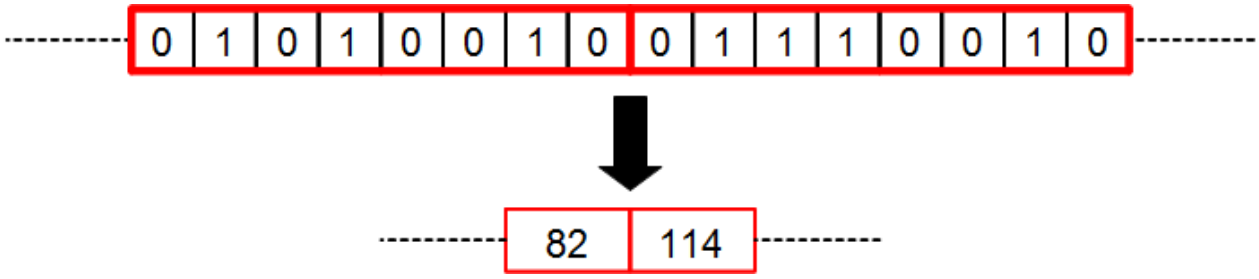
Or, en pratique, la mémoire de l'ordinateur peut se voir comme un long ruban, constitués de *bits* qui valent 0 ou 1, un peu comme cela :



Les *bits* sont en pratique arrangés par blocs de huit, qu'on appelle *octets*. Un bloc de huit bits, par exemple 01010010 peut se voir comme l'écriture en base 2 d'un entier :

$$(0|1|0|1|0|0|1|0) = 0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 82.$$

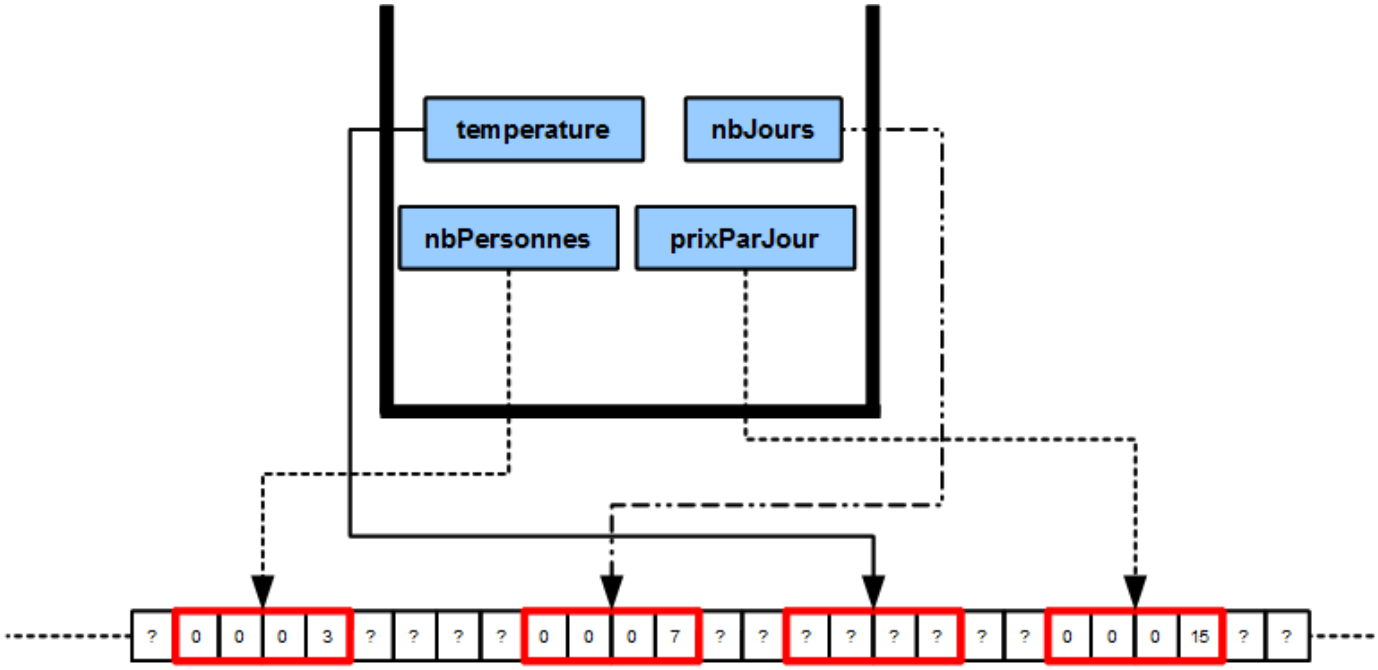
La mémoire ressemble alors à une suite d'octets :



Un entier, lui, correspond à 32 bits donc à 4 octets, par exemple :

$$(100|30|240|25) = 100 \times 256^3 + 30 \times 256^2 + 240 \times 256^1 + 25 \times 256^0 = 1679749145$$

Ainsi, on peut représenter chaque variable en reliant son nom à l'emplacement en mémoire où elle est stockée :



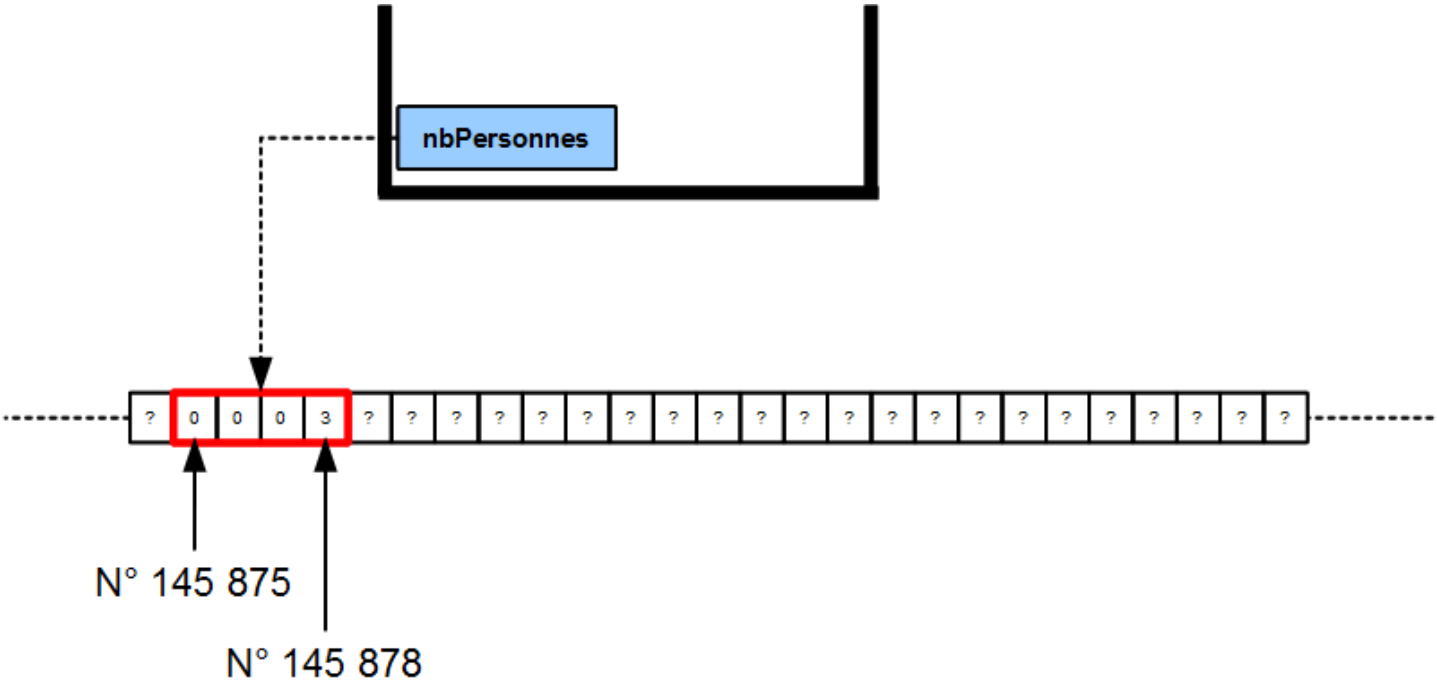
Remarque : on s'est placé ici dans le cadre d'une architecture à 32 bits. Or, de plus en plus d'ordinateurs fonctionnent avec une architecture 64 bits, dans laquelle on utilise 8 octets (donc 64 bits) au lieu de 4 octets (32 bits). Le principe étant exactement le même, nous continuerons à présenter des exemples en 32 bits, ce qui permet d'avoir des illustrations plus faciles à lire.

Adresse d'une variable

À chaque variable on peut associer une longueur en mémoire (toujours égale à 4 octets pour une variable entière) ainsi que le numéro du premier octet de la mémoire où le contenu de la variable est présent. On appelle ce numéro l'*adresse de la variable*. Sur l'exemple de code suivant

```
int nbPersonnes = 3;
```

on pourrait par exemple avoir la situation suivante :

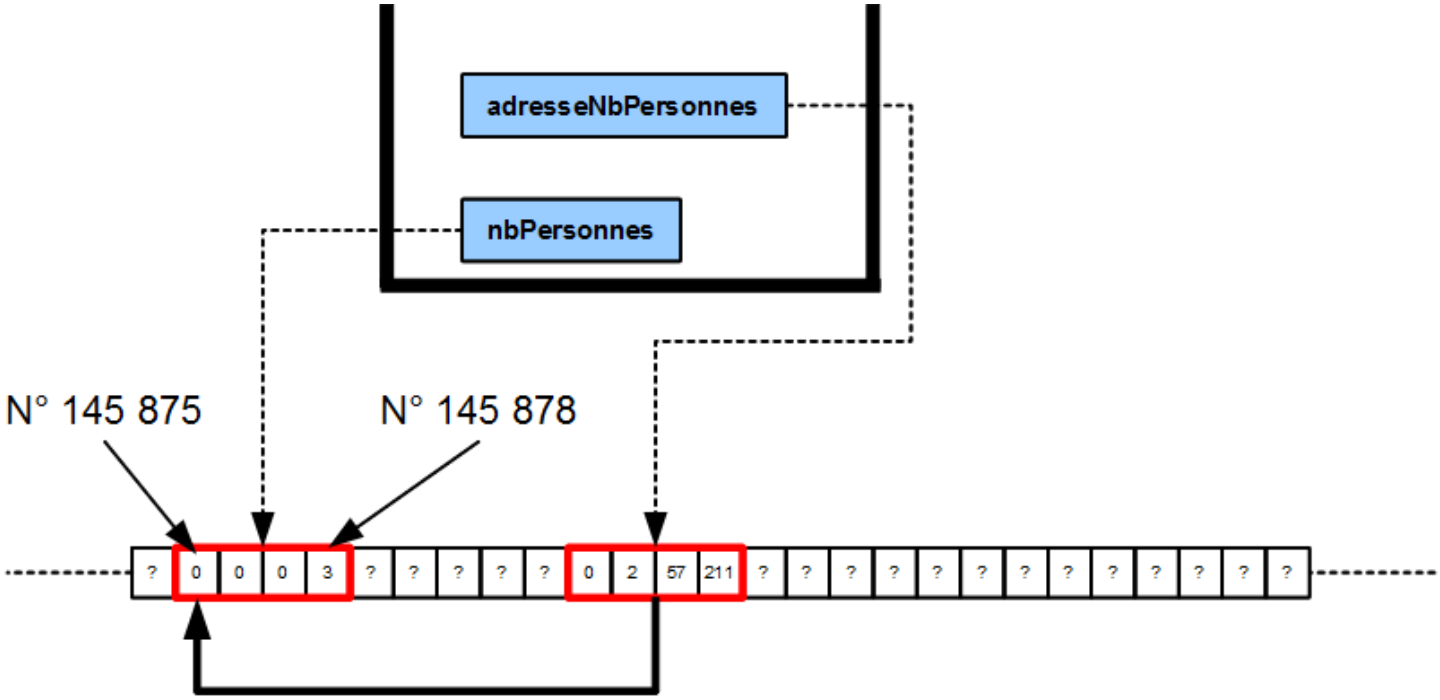


Nous avons donc une variable `nbPersonnes` qui commencent à l'octet numéro 145 875 et finit au numéro 145 878.

Eh bien, en C++, il est possible de connaître l'adresse d'une variable ! Regardons un petit extrait de code et la représentation graphique associée, nous pourrons alors analyser ce qui se passe.


```
int nbPersonnes = 3;
```

```
int* adresseNbPersonnes = &nbPersonnes;
```



La variable `adresseNbPersonnes` contient donc la valeur 145 875, car

$$(0|2|57|211) = 0 \times 256^3 + 2 \times 256^2 + 57 \times 256^1 + 211 \times 256^0 = 145875,$$

et on a ajouté une flèche partant de son espace mémoire vers celui de la variable `nbPersonnes` pour bien montrer qu'il "pointe" vers ce second espace mémoire.

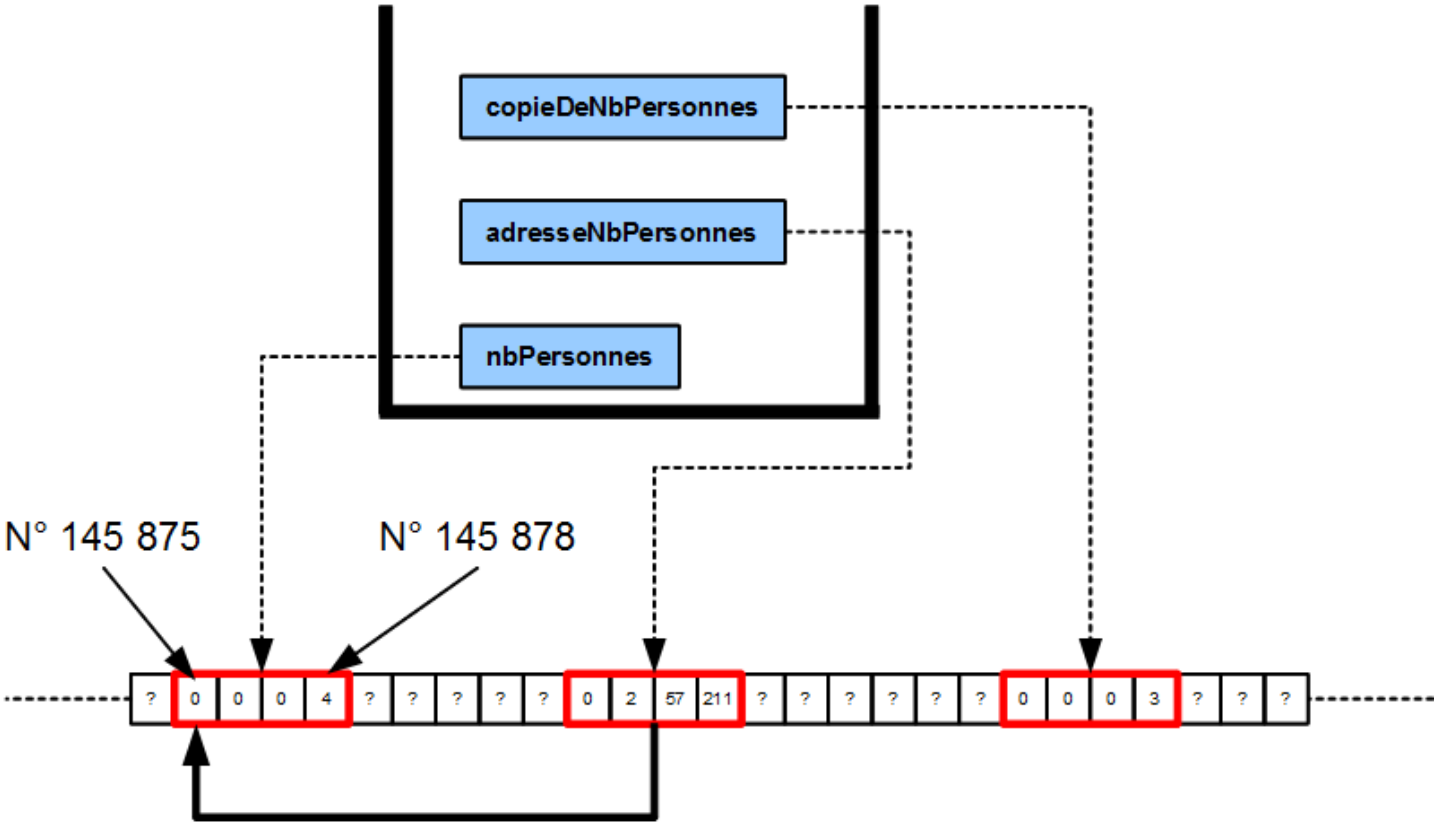
En terme de code, on a utilisé l'opérateur `&` qui permet de récupérer l'adresse d'une variable (ici `nbPersonnes`) et de la stocker dans une autre variable (ici `adresseNbPersonnes`) qui est de type "contient l'adresse d'une variable de type entier" ou de manière plus courte "*pointeur* vers un entier" et on note ce type `int*`.

Bilan Partiel : chaque variable est représentée en mémoire par une suite de case consécutives et il est possible de connaître le numéro de la première case, qu'on appelle adresse de la variable.

Valeur à partir de l'adresse

Nous venons de voir comment récupérer l'adresse d'une variable et la stocker quelque part, mais si on nous donne une adresse, peut-on connaître la valeur qui est stocké à cette adresse ? Bien sur que oui ! On précède ainsi :

```
int nbPersonnes = 3;
int* adresseNbPersonnes = &nbPersonnes;
int copieDeNbPersonnes = *adresseNbPersonnes;
nbPersonnes = 4;
```



À la ligne 3, on est allé chercher la valeur stockée à l'adresse indiquée dans la variable `adresseNbPersonnes`. Cette valeur valait alors 3 et on l'a stockée dans la variable `copieDeNbPersonnes`. On a ensuite modifié la valeur de `nbPersonnes`, ce qui n'a pas changé la valeur de `copieDeNbPersonnes`, car il s'agit bien de deux espaces différents en mémoire.

Espace mémoire à partir de l'adresse

Précédemment, lorsque nous avons placé le code `*adresseNbPersonne` à droite du signe `=`, on a récupéré une valeur, celle stockée à l'adresse correspondante. Mais il est aussi possible de placer ce code à gauche du signe `=` :

```
int nbPersonnes = 3;
int* adresseNbPersonnes = &nbPersonnes;
*adresseNbPersonnes = 4;
cout << nbPersonnes;
```

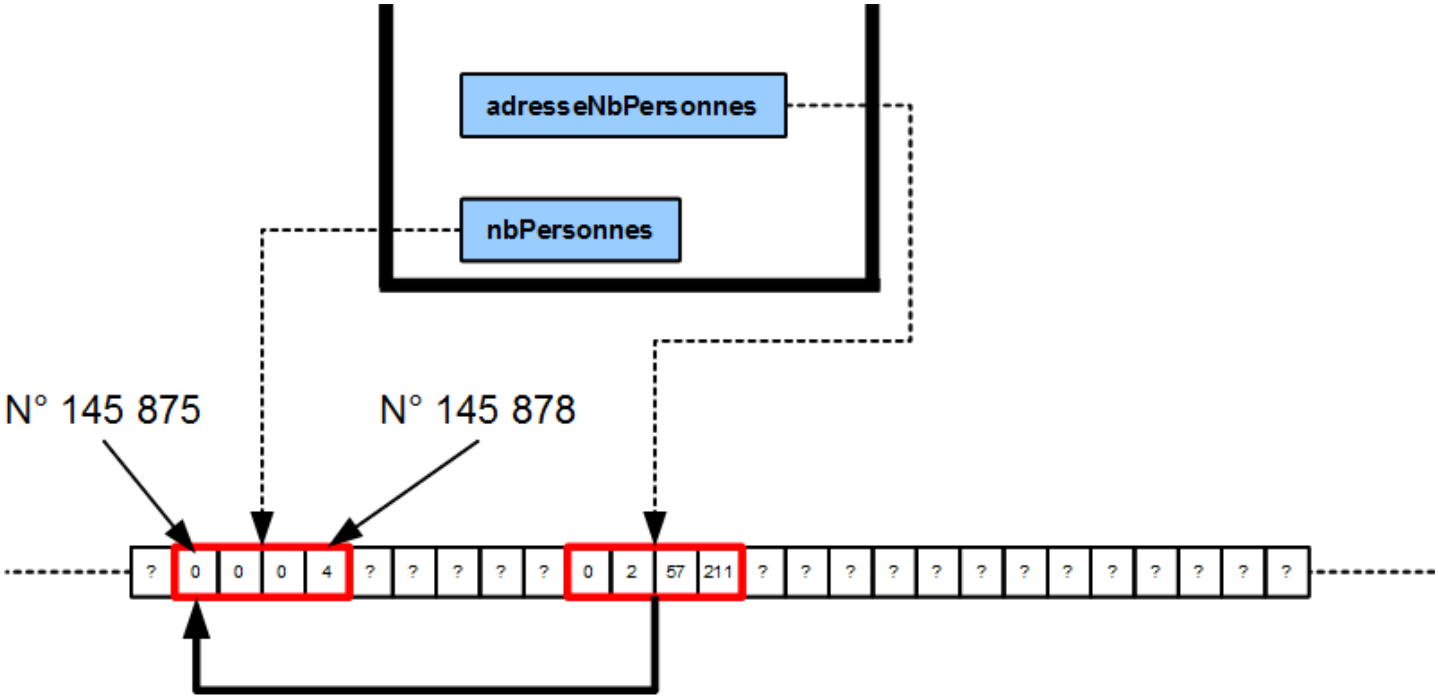
↳
```
4
```

La troisième ligne, c'est-à-dire

```
*adresseNbPersonnes = 4;
```

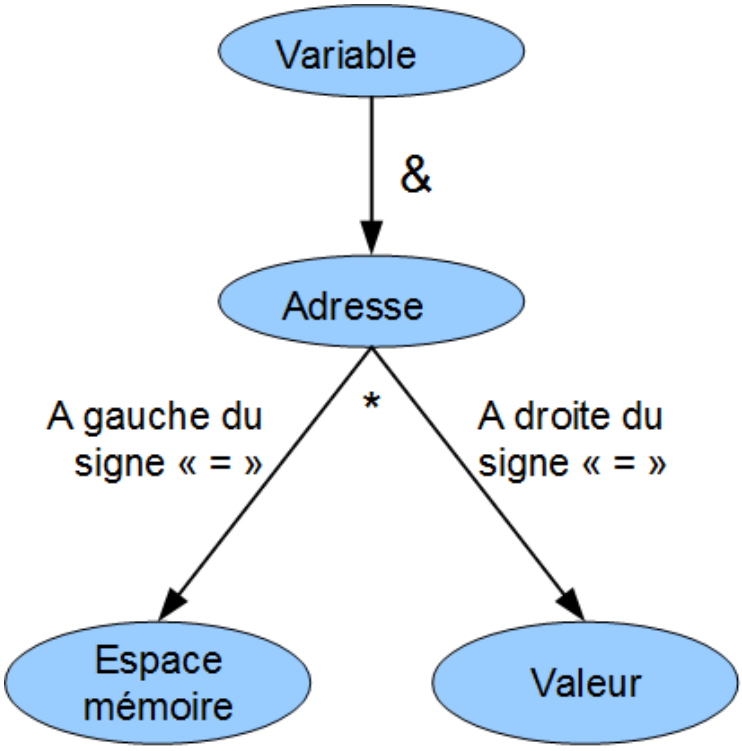
signifie : mettre la valeur 4 dans l'emplacement mémoire dont l'adresse est contenu dans la variable `adresseNbPersonnes`.

En mémoire, on a donc :



Lien entre adresse et valeurs

On peut donc résumer les choses ainsi :



- À partir d'une variable, on utilise l'opérateur `&` pour accéder à son adresse.
- À partir d'une adresse, et si on est à droite du signe `=`, on utilise l'opérateur `*` pour accéder à la valeur située dans l'espace mémoire vers lequel l'adresse pointe.
- À partir d'une adresse, et si on est à gauche du signe `=`, on utilise l'opérateur `*` pour accéder à l'espace mémoire vers lequel l'adresse pointe.

En y réfléchissant bien, une variable se comporte exactement de la même manière : à gauche du signe `=` c'est un emplacement de la mémoire et à droite du signe `=` c'est une valeur. Ainsi, étant donné une adresse mémoire `monAdresse`, on pourra manipuler `*monAdresse` de la même manière qu'on manipulerait une variable.

Par exemple, les deux codes suivants sont équivalents :

```
int* b = &a;

*b = *b + 6;
*b = *b * c;
```

```
a = a + 6;
a = a * c;
```

On voit bien que `*b` se manipule exactement comme on manipule `a`.

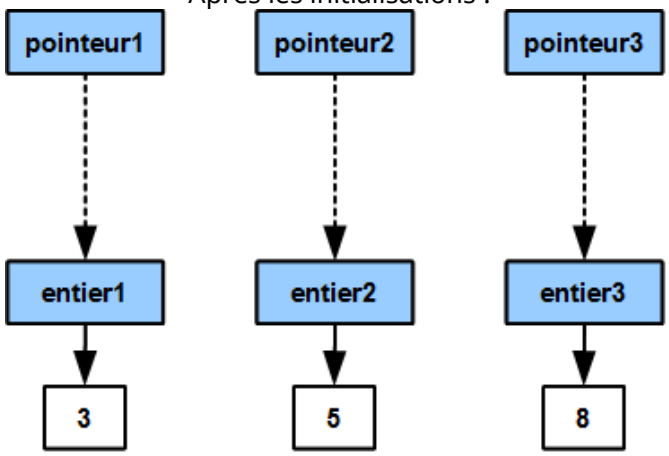
Quelques manipulations

Voici un petit extrait de programme, dont les variables ont volontairement des noms non-explicites. Vous devez essayer de déterminer la valeur de chaque variable à la fin du programme, soit une valeur entière, soit une flèche vers une autre variable. Faites des dessins !

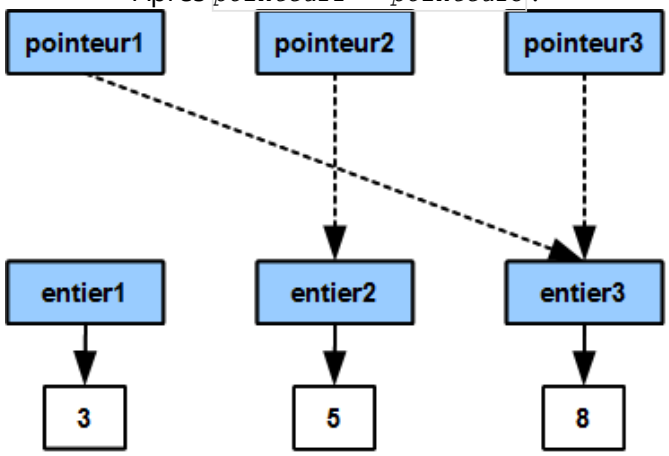
```
int entier1 = 3;
int entier2 = 5;
int entier3 = 8;
int* pointeur1 = &entier1;
int* pointeur2 = &entier2;
int* pointeur3 = &entier3;

pointeur1 = pointeur3;
*pointeur3 = 10;
entier2 = *pointeur1 + 5;
```

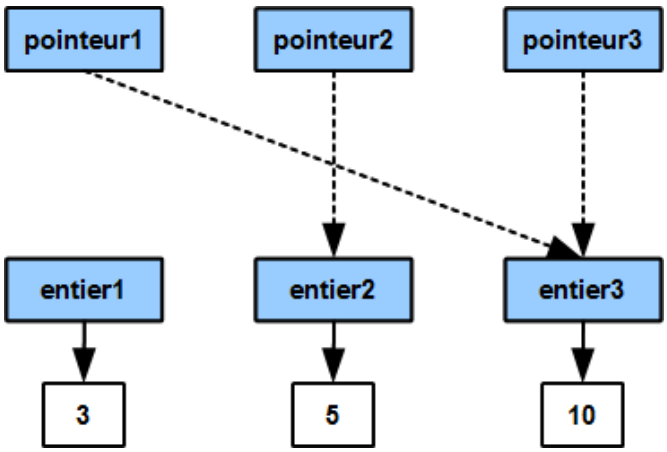
Après les initialisations :



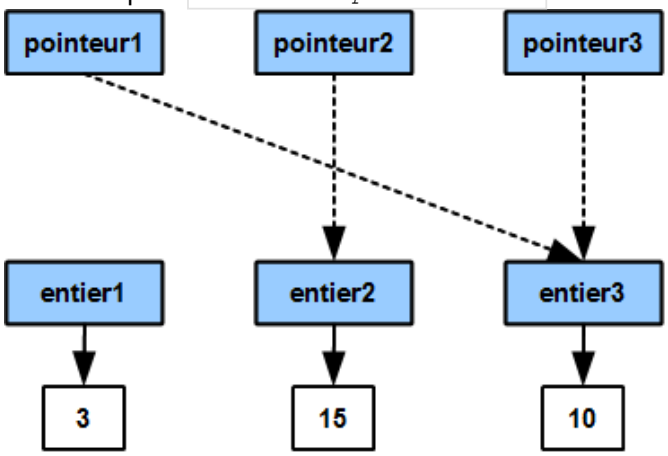
Après `pointeur1 = pointeur3` :



Après `*pointeur3 = 10` :



Après `entier2 = *pointeur1 + 5` :



Un peu de vocabulaire

Dans ce cours, nous vous avons présenté deux nouveaux opérateurs, `&` et `*`, voici leur noms, que nous ne vous demandons pas de retenir.

- L'opérateur `&` s'appelle *l'opérateur d'indirection*
- L'opérateur `*` s'appelle *l'opérateur de déréférencement*

Conclusion

Voilà ce petit cours sur la mémoire et les pointeurs terminés. Il nous permettra de vous présenter plus facilement certains prochains cours sur les tableaux ou les chaînes de caractères. Ne vous inquiétez pas si tout n'est pas très clair, c'est un sujet difficile. Essayez de retenir les schémas présentés et les notions d'adresses et de pointeurs.

En pratique, à l'exception de quelques cas très précis que nous vous présenterons, nous n'utiliserons jamais directement des pointeurs au sein des corrections des exercices et nous vous conseillons d'en utiliser le moins possible. Ils sont en effet un petit peu difficiles à manipuler et source de nombreuses erreurs.