Information, Calcul et Communication

Composante Pratique: Programmation C++

MOOC sem7 : pointeur (1)

Adresse d'une variable

Différences entre référence et pointeur

Le bon usage d'un pointeur

La semaine prochaine: allocation dynamique et pointeur

Adresse d'une variable (rappel cours S2)

Un processeur est conçu pour que ses instructions travaillent sur un nombre prédéfini d'octets, appelé un mot :

- 4 octets : machine 32 bits
- 8 octets : machine 64 bits

La mémoire centrale est aussi organisé en mots de même taille que le processeur.

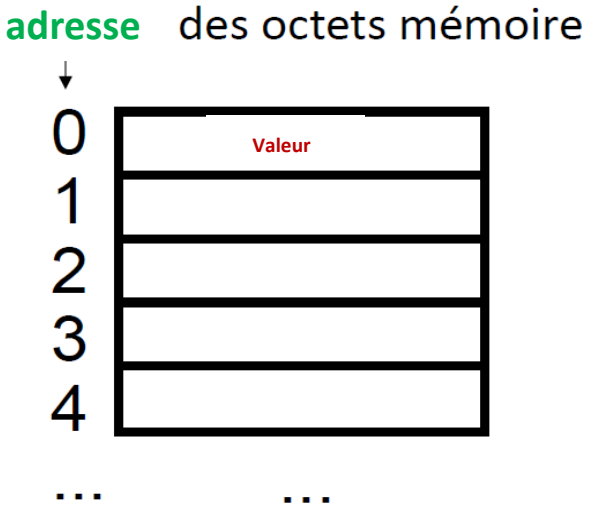
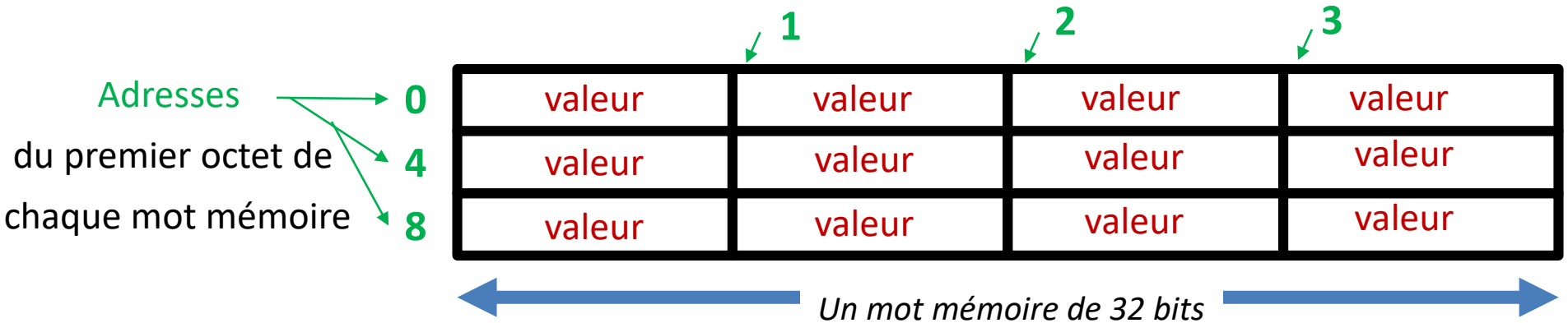


Illustration pour ***une machine 32 bits***



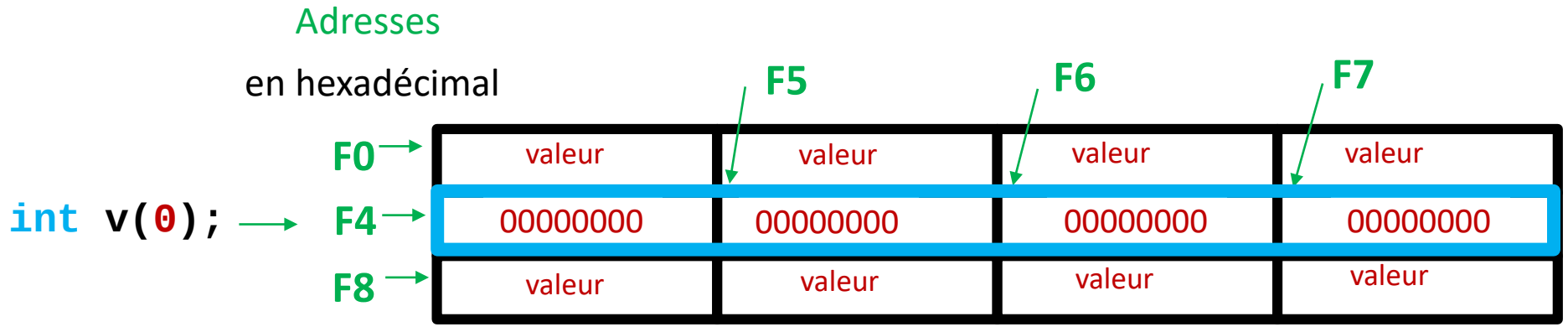
Adresse d'une variable

Le compilateur réserve 1 ou plusieurs octets pour mémoriser la valeur d'une variable selon son type.

Ex: 1 octet pour **bool** et pour **char**, 4 octets pour **int**, 8 octets pour **double**

L'adresse d'une variable est l'adresse du premier octet réservé pour mémoriser la valeur de cette variable.

Une adresse étant aussi un motif binaire, on l'écrit sous forme condensée en base 16.
En C++ on peut manipuler des constantes en hexadécimal en indiquant **`0x`** avant sa valeur. Ex: **`0xF0`**



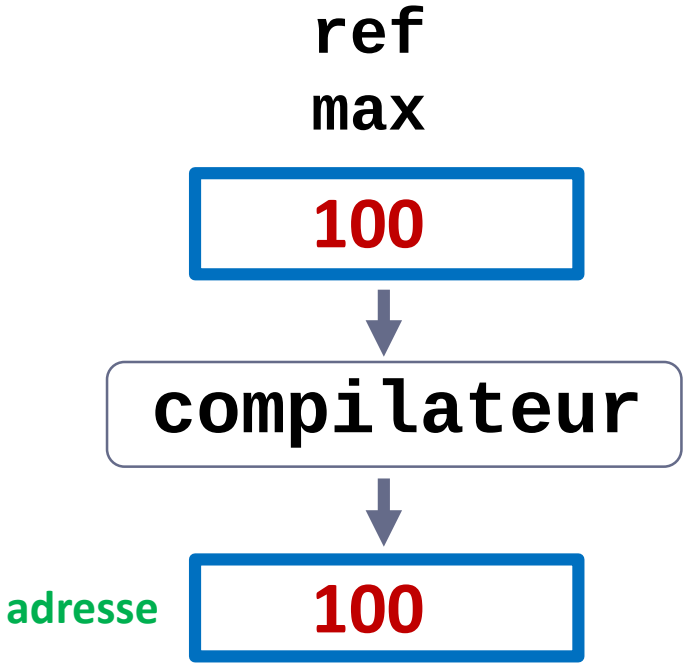
L'adresse de la variable **v** est F4

machine 32 bits

Différence entre référence (vu en S5) et pointeur

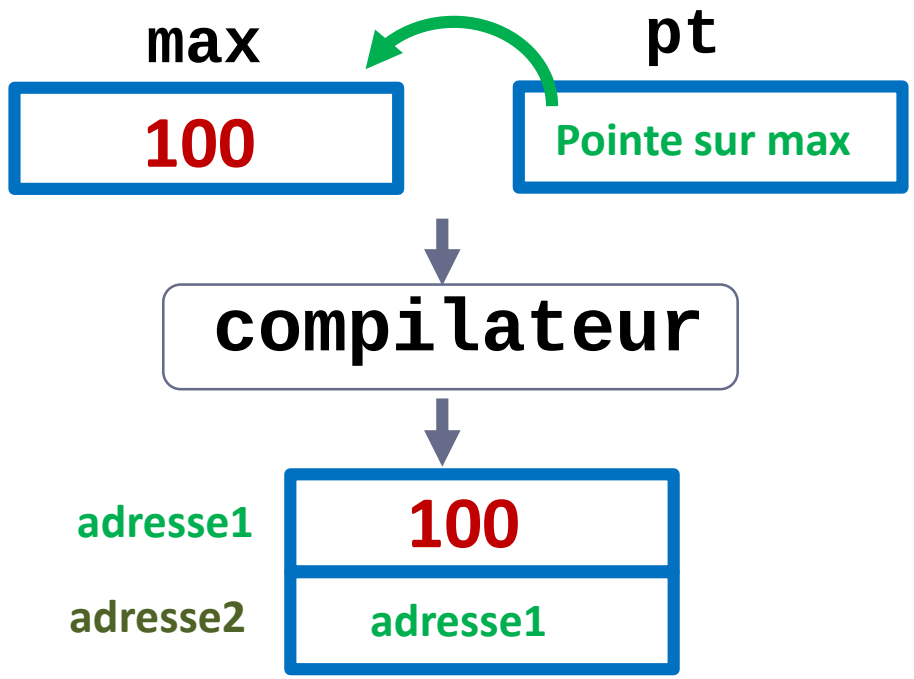
Une référence est un second nom permanent et invariant

```
int max(100);
int &ref = max;
```



Un pointeur est une **variable** indépendante qui mémorise une adresse vers un type bien défini de donnée (ici vers un **int**)

```
int max(100);
int *pt = &max;
```

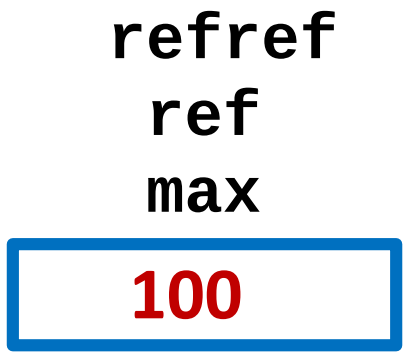


// une adresse occupe 4 octets sur une machine 32 bits
// une adresse occupe 8 octets sur une machine 64 bits

Référence de référence / Pointeur de pointeur

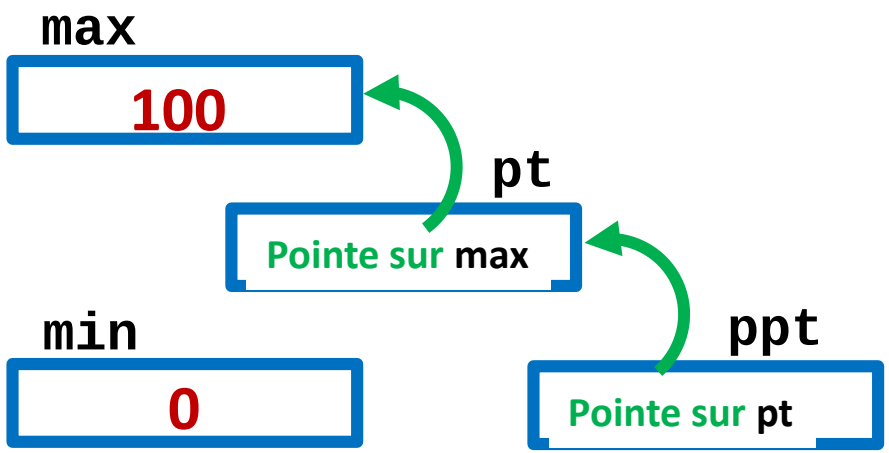
Il est possible de déclarer une référence sur une référence: par transitivité la *référence de référence* devient simplement un troisième nom de la variable référencée.

```
int max(100);
int &ref = max;
int &refref = ref;
```



La déclaration et l'initialisation d'un pointeur de pointeur sont plus délicates : il faut rester cohérent concernant le type de l'objet pointé.

```
int max(100),min(0);
int *pt   = &max;
int **ppt = &pt;
```




```
*ppt = &min;
```
// que se passe-t-il ?

Equivalence entre pointeur à-la-C et tableau à-la-C

pointeur_tab_a_la_C.cc

La déclaration suivante d'un pointeur montre deux informations importantes au compilateur :

```
int* p;
```

1) l'étoile * montre que **p** mémorise une adresse
2) Le type **int** permet au compilateur de savoir que 4 octets sont associés à l'adresse mémorisée par **p**.

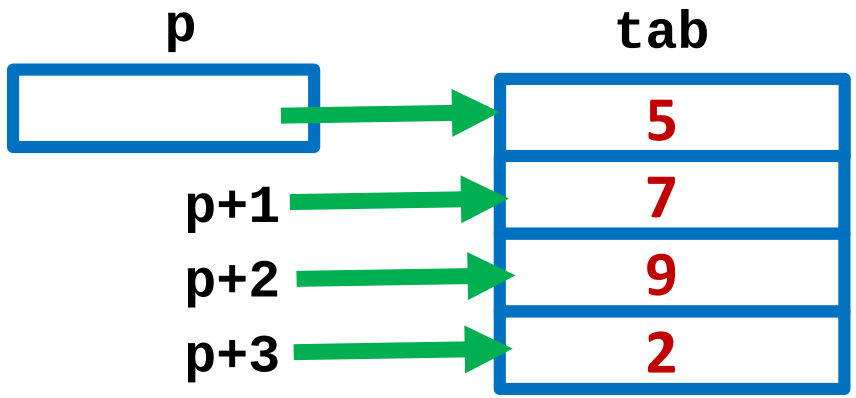
L'expression **p + 1**

veut dire : **p + 1*(*taille de l'objet pointé*)octets**

Cette propriété est particulièrement utile quand un pointeur **p** travaille avec un tableau à-la-C **tab** comme suit :

```
int tab[4]={5,7,9,2};
int* p = tab;
```

*Le nom **tab** est un pointeur constant sur le **premier élément** du tableau*



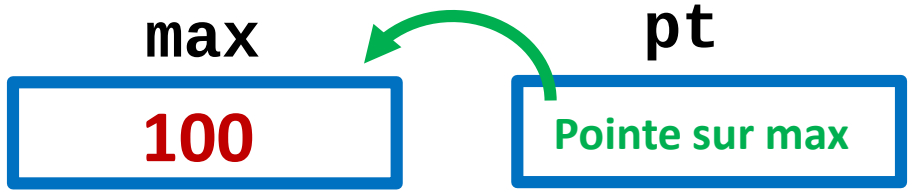
Ensuite on peut utiliser **p** pour accéder et manipuler tous les éléments de **tab** car **p+i** est équivalent à: **&tab[i]**

Le bon usage d'un pointeur

Une référence quand on peut, un pointeur quand on doit.

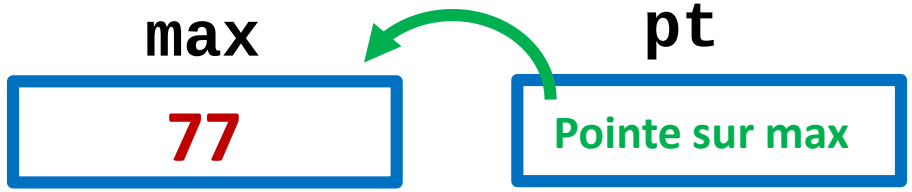
1) Un pointeur DOIT être initialisé avec une adresse valide AVANT d'être utilisé pour lire ou écrire en mémoire avec l'opérateur *.

```
int max(100);
int *pt = &max;
```



Si le pointeur est destiné à toujours pointer sur la même variable, autant utiliser une **référence** car sa syntaxe est plus lisible et elle présente moins de risques.

```
// avant
*pt = 77;
// après
```

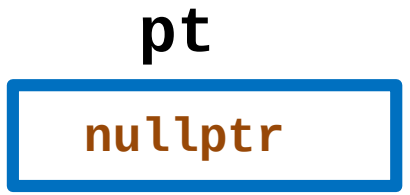


Par exemple: le **passage par référence** DOIT être préféré au passage de la valeur d'un pointeur pour modifier une variable externe à la fonction.

Le bon usage d'un pointeur (2)

2) Si au moment de la déclaration on ne sait pas encore sur quoi doit pointer un pointeur, alors il FAUT l'initialiser avec la valeur **nullptr** qui est équivalente à **false**.

```
int *pt(nullptr);
```



Cela veut dire que ce pointeur est «**désactivé**» et qu'il ne DOIT PAS être utilisé pour lire ou écrire en mémoire avec *.

On obtiendrait segmentation fault à l'exécution de l'expression: ***ptr**

Le bon usage d'un pointeur (3)

3) Cas général: la valeur d'un pointeur pouvant être modifiée pendant l'exécution du programme, il FAUT :

3.1) toujours tester s'il est différent de **nullptr** AVANT de lire ou écrire en mémoire avec *

3.2) toujours désactiver un pointeur en lui affectant la valeur **nullptr** s'il doit temporairement ne plus être utilisé

```
int *pt(nullptr);
…
if(pt != nullptr)
{
    … *pt … ;
}
if(pt) // écriture équivalente :
{
    …
    *pt = … ;
    …
    pt = nullptr;
}
```