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.

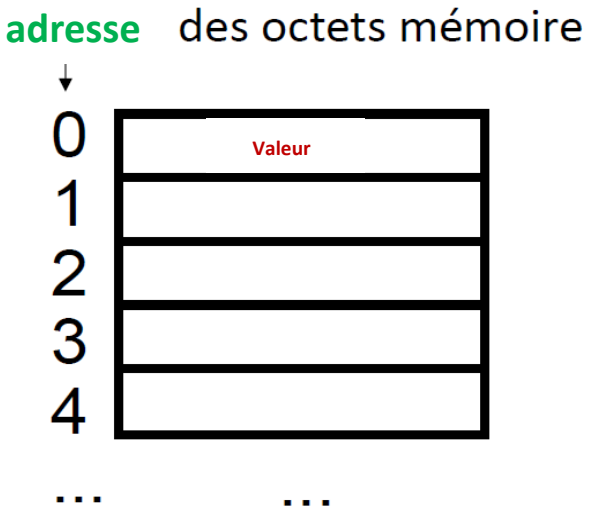
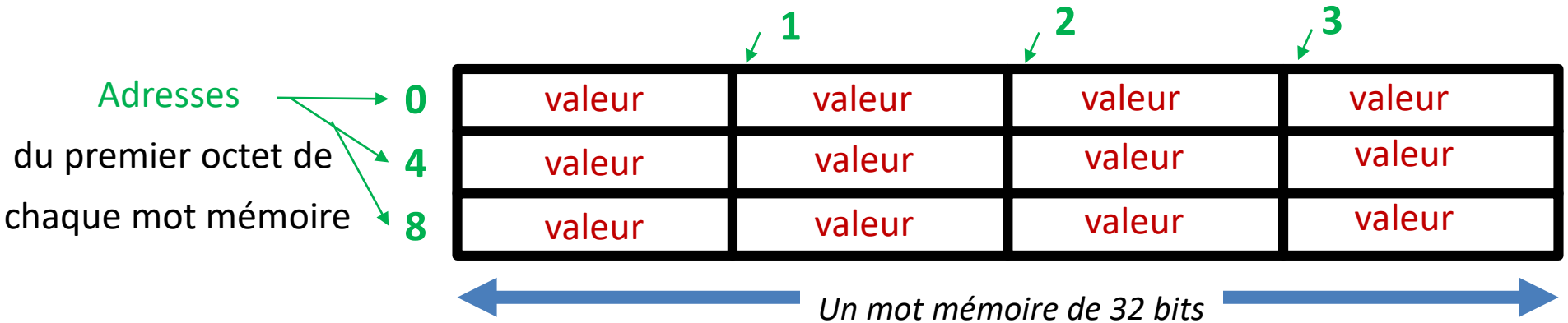


Illustration pour ***une machine 32 bits***



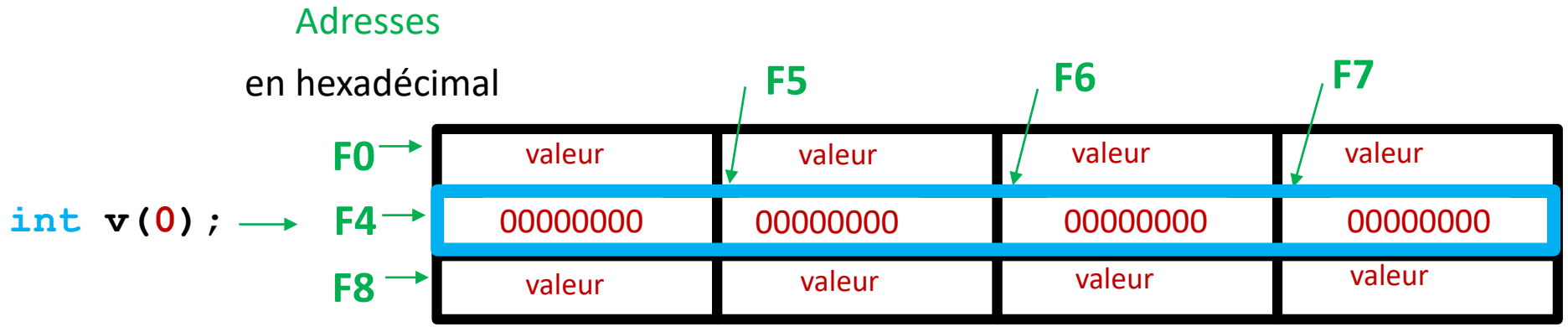
Adresse d'une variable

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 **0xF4**

Premiers opérateurs liés aux pointeurs

Accès à un élément d'un tableau

Indirection/déréférencement

Calcul de l'adresse d'une variable

sur 17 niveaux de priorités

Associativité: pour les opérateurs de même priorité

Gauche->Droite / Left-to-Right
Droite->Gauche / Right-to-Left

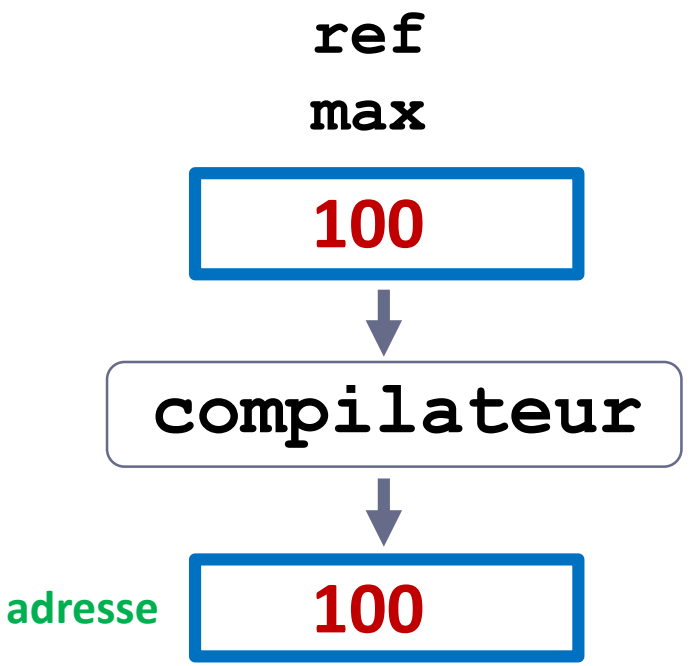
Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	a++ a--	Suffix/postfix increment and decrement	
	type() *type*{}	Functional cast	
	a()	Function call	
	a[]	Subscript	
	. ->	Member access	
3	++a --a	Prefix increment and decrement	Right-to-left
	+a -a	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	C-style cast	
	*a	Indirection (dereference)	
	&a	Address-of	
	sizeof	Size-of[note 1]	
	co_await	await-expression (C++20)	
	new new[]	Dynamic memory allocation	
	delete delete[]	Dynamic memory deallocation	
4	.* ->*	Pointer-to-member	Left-to-right
5	a*b a/b a%b	Multiplication, division, and remainder	
6	a+b a-b	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	
8	<=>	Three-way comparison operator (since C++20)	
9	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
10	== !=	For relational operators = and ≠ respectively	
11	&	Bitwise AND	
12	^	Bitwise XOR (exclusive or)	
13	|	Bitwise OR (inclusive or)	
14	&&	Logical AND	
15	||	Logical OR	
16	a?b:c	Ternary conditional[note 2]	Right-to-left
	throw	throw operator	
	co_yield	yield-expression (C++20)	
	=	Direct assignment (provided by default for C++ classes)	
	+= -=	Compound assignment by sum and difference	
	*= /= %=	Compound assignment by product, quotient, and remainder	
	<<= >>=	Compound assignment by bitwise left shift and right shift	
	&= ^= |=	Compound assignment by bitwise AND, XOR, and OR	
17	,	Comma	Left-to-right

cppreference.com

Différence entre référence (vu en S5) et pointeur

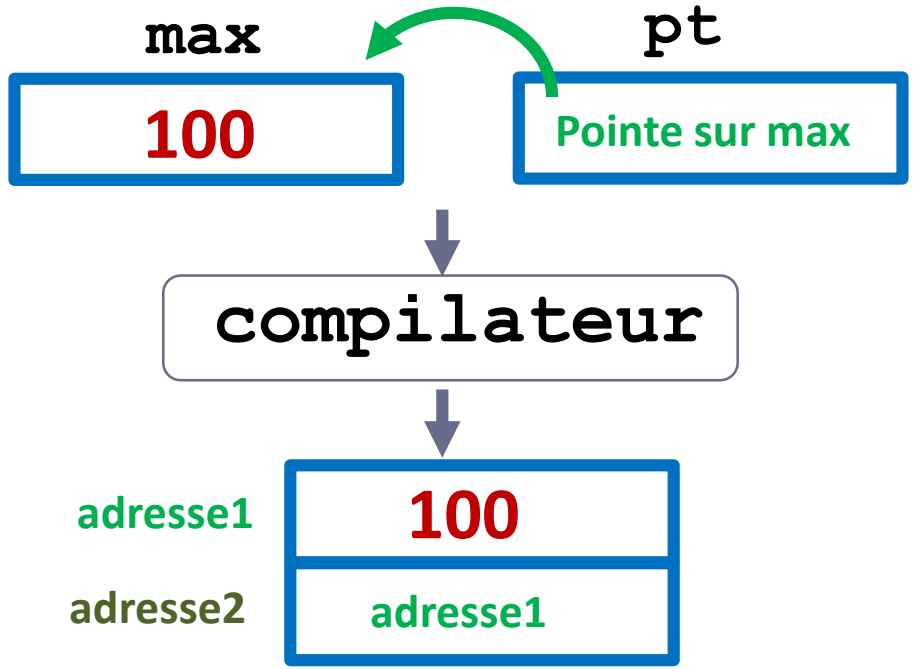
Une référence est un second nom permanent et invariant

```
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 jusqu'à 8 octets sur une machine 64 bits

pointeur_vs_ref.cc

Rappel: En C++ une variable possède un ***type*** qui détermine le nombre d'octets qu'elle occupe en mémoire pour stocker sa ***valeur***.

L' ***adresse*** d'une variable est l'adresse de son premier octet

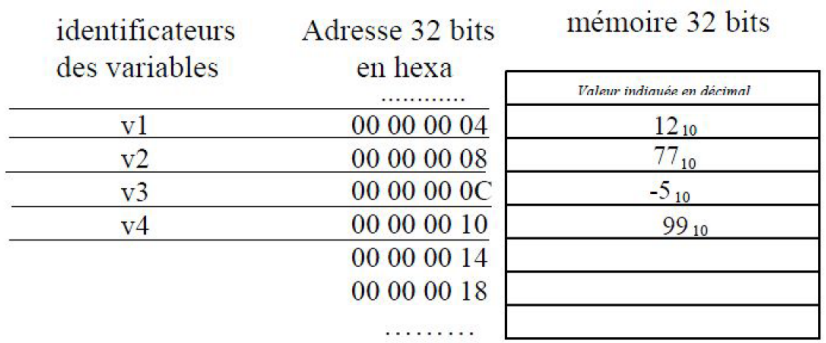
Grâce à ***l'identificateur*** de la variable (son nom) on peut lire sa valeur ou la modifier sans connaître explicitement son adresse. En effet ce sont des traitements ultérieurs à l'écriture du programme qui décident de sa position en mémoire au moment de l'exécution. Malgré cela il est très important de maîtriser le concept d'adresse pour la suite du cours.

A droite d'un opérateur '=' l'***identificateur*** est équivalent à ***"la valeur de la variable"***.

Le langage C++ définit par ailleurs les 2 opérateurs **&** et *, comme suit :

& ***identificateur*** est équivalent à l' "***adresse*** *de la variable* ***identificateur***".
***** ***adresse*** est équivalent à la "***valeur*** *dans la mémoire à l'adresse* ***adresse***".

Dans l'illustration suivante, nous avons quatre variables de type **int** d'identificateurs : v1, v2, v3, v4 . Pour simplifier, la valeur **int** en mémoire est en décimal bien qu'elle ne soit en réalité qu'une suite de 0 et de 1.



identificateurs des variables	Adresse 32 bits en hexa	mémoire 32 bits
	.....	*Valeur indiquée en décimal*
v1	00 00 00 04	12_{10}
v2	00 00 00 08	77_{10}
v3	00 00 00 0C	-5_{10}
v4	00 00 00 10	99_{10}
	00 00 00 14	
	00 00 00 18	
	.....	

Questions	Réponses: **int** *en decimal, adresse en hexa avec 0x*
1) valeur de **v1, v2, v3, v4**	12, 77, -5, 99
2) adresse de **v1** (en hexa)	0x4
3) **&v3**	0xC
4) valeur en mémoire à l'adresse **&v3**	-5
5) ***(&v4)**	99
Attention ces expressions ne sont pas en C++ ; voir le slide sur l'arithmétique sur les pointeurs et les tableaux à-la-C pour toute addition avec un pointeur.	
6) **&v1 + 4 octets**	0X4
7) ***(&v1 + 8 octets)**	-5

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**

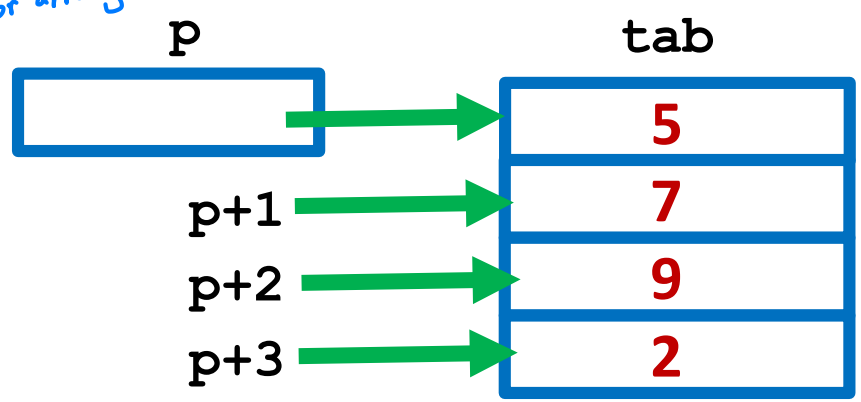
Cette propriété est particulièrement utile quand un pointeur **p** travaille avec un tableau à-la-C **tab** comme suit :

```
int tab[4]={5,7,9,2};
int* p = tab; // &tab[0];
```

Only for C-tab won't work for array or vector.

to for vector or array

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



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

Un pointeur est une variable dont la valeur est une adresse mémoire. Plus précisément, la déclaration d'une variable pointeur fait apparaitre le type de l'objet pointé (ex: adresse d'un int, adresse d'un float, etc...).

Un tableau-à-la-C occupe un bloc compact d'espace mémoire suffisant pour contenir la valeur de tous ses éléments.

Contrairement aux variables (classique ou pointeur) **le nom du tableau-à la-C, utilisé seul dans une expression, est une constante**: c'est l'adresse du premier élément du tableau.

Dans l'illustration suivante, nous avons des variables de type **`int`**, des variables pointeurs de type **`int*`** et un tableau-à-la-C de 2 **`int`**.
Pour simplifier vous indiquerez les valeurs **`int`** en décimal et les adresses en hexadécimal.
Complétez le contenu de la mémoire après les déclarations
et instructions ci-dessous et ensuite donnez la valeur des expressions apparaissant dans les questions de 1) à 10). Mettez à jour l'état de la mémoire si nécessaire.

Illustrez aussi l'exercice en utilisant les conventions graphiques « boite + flèche »

```
int v1 = 19;
int v2 = 50 ;
int* v3 = &v1 ;
int* v4 = &v2 ;
int  v5[2] = {-13, 27};
```

déclarations		Adresse en hexa
		....
int	v1	04
int	v2	08
int*	v3	0C
int*	v4	10
int	v5[2]	14
		18

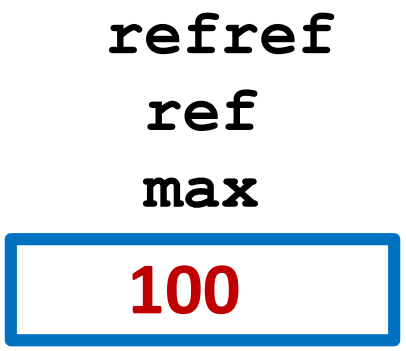
mémoire 32 bits

Questions	Réponses: *valeur* **int** *en decimal, adresse en hexa avec 0x*
1) valeur de l'expression: **v1**	19
2) valeur de l'expression: **v3**	0x4
3) valeur de: ***v3** = **v3[0]**	19
4) valeur de l'expression: **&v3**	0xC
5) valeur de l'expression: **v5**	0x14
6) valeur de l'expression: **v5[0]**	-13
7) valeur de l'expression: **&v5[1]**	0x18
8) valeur de l'expression: ***v4 + 2**	52
9) valeur de l'expression: **(*v4)++** *quelque chose change en mémoire*	50 c'est une post incrémentation puis +1
10) valeur de l'expression: ***v3= *v4 + 3** *Prendre en compte l'action de 9)*	51+3=54 → V1 sera = 54

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.

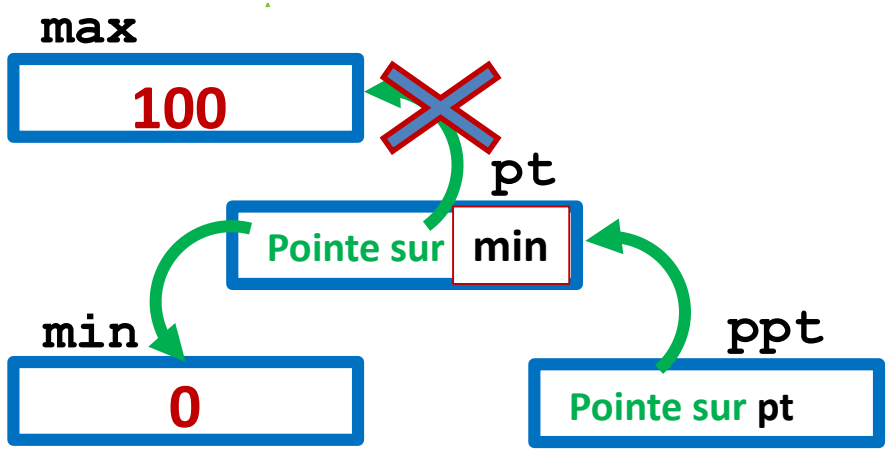
```
int max(100);
int &ref = max;
int &refref = ref;
```



reference_de_reference.cc

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 ?
```

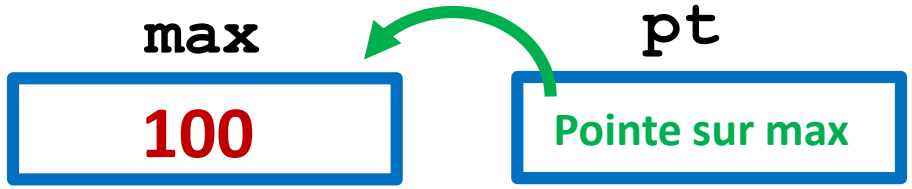
pointeur_de_pointeur.cc

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;
```



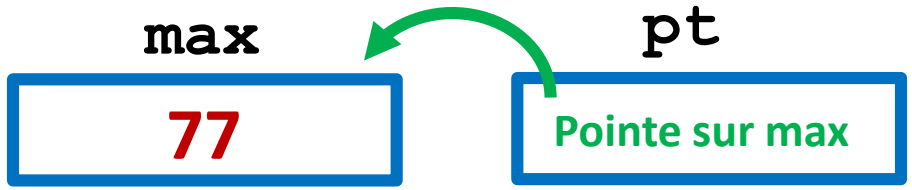
Si le pointeur est destiné à toujours pointer sur la même variable, autant utiliser une **référence** car sa syntaxe est plus lisible et elle présente moins de risques.

// avant

```
*pt = 77;
```

// après

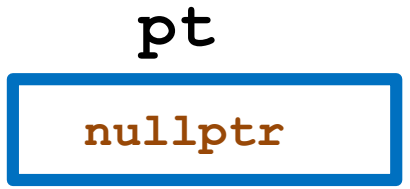
Par exemple: le **passage par référence** DOIT être préféré au passage de la valeur d'un pointeur pour modifier une variable externe à la fonction.



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;
}
```