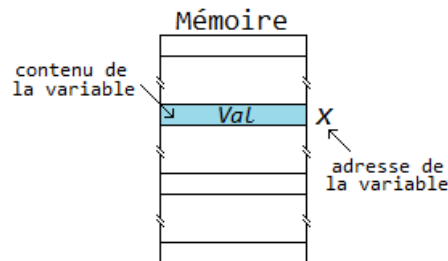


Liste d'exercices 7 : les pointeurs (1 séance)

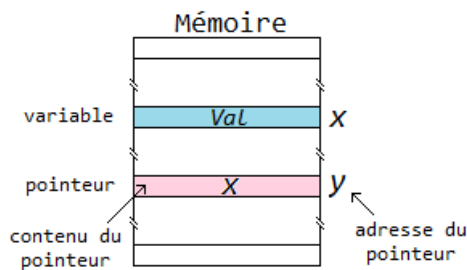
Notions à considérer pour pouvoir faire les exercices

Introduction aux pointeurs :

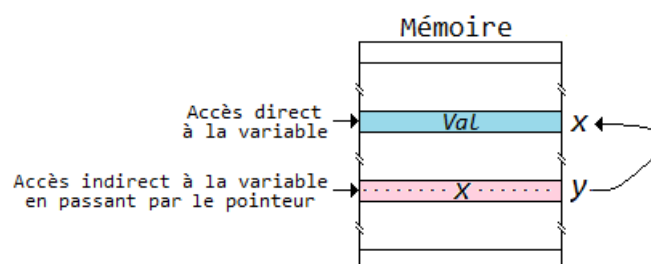
Une variable est localisée à une adresse précise en mémoire et elle stocke une valeur. Dans le diagramme suivant de la mémoire, la variable est localisée à l'adresse x et elle stocke la valeur Val .



En C, un pointeur est une variable qui mémorise non pas une valeur, mais l'adresse d'une autre variable. Étant donné qu'une **adresse est codée sur 4 octets**, un pointeur occupe 4 octets en mémoire. Dans le diagramme suivant de la mémoire, le pointeur est localisé à l'adresse y et il stocke l'adresse x qui est l'adresse de la variable d'origine.



On peut à présent accéder de 2 façons différentes à la variable d'origine : soit **directement** en mentionnant son nom, soit **indirectement** en passant par le pointeur.



Les opérateurs & et * pour les pointeurs :

En C, les opérateurs & et * relatifs aux pointeurs sont des opérateurs unaires qui ont la même priorité que les autres opérateurs unaires.

<i>Priorités</i>	<i>Opérateurs en C</i>
+++++++++	()
+++++++++	~, -, &, *, (casting), (changement de nature)
+++++++++	*, /, %
+++++++	+, -
++++++	<<, >>
+++++	&
+++	^
++	
+	=

- **L'opérateur unaire &** permet d'obtenir l'adresse d'une variable. En assembleur, on utilise OFFSET dans l'opérande de l'instruction.

Exemple : `pi = &i;` correspond à `MOV pi, OFFSET i`

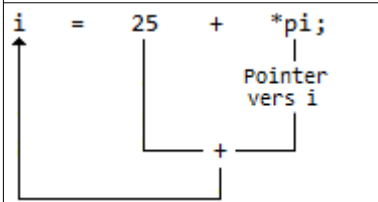
- **L'opérateur unaire *** est placé devant un pointeur pour accéder indirectement, c'est-à-dire en passant par le pointeur, à la variable dont l'adresse se trouve dans ce pointeur.

Les pointeurs en assembleur :

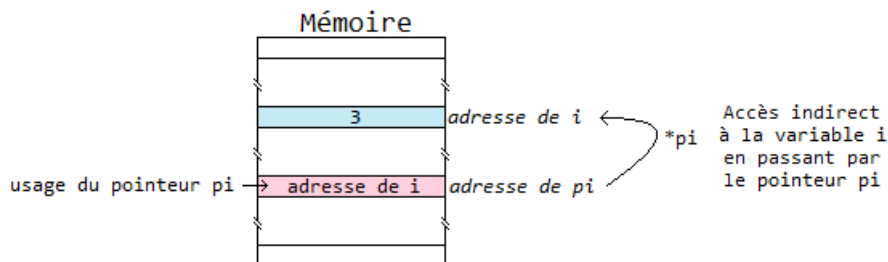
Pour le processeur, les accès indirects en mémoire se font uniquement via les registres généraux: EAX, EBX, ECX, EDX, ESI ou EDI, utilisés comme pointeurs, peu importe le type des variables : char, short, int, float ou double.

Exemple: accéder indirectement à une variable du type int :

```
int i = 3, *pi = &i;
```

<i>Langage C</i>	<i>Assembleur</i>
 <pre>i = 25 + *pi;</pre>	<pre>mov eax, 25 mov ebx, pi add eax, DWORD PTR [ebx] mov i, eax</pre>

En C, quand on utilise `*pi`, on atteint la variable `i` en passant par le pointeur `pi` :



En assembleur, pour réaliser une opération similaire, on doit passer par un registre général ainsi :

```
mov ebx, pi          (1)
add eax, DWORD PTR [ebx] (2)
```

L'instruction (1) copie dans EBX l'adresse de la variable `i` se trouvant dans le pointeur `pi`. L'instruction (2) passe par EBX utilisé comme pointeur pour récupérer la valeur de la variable `i` afin de l'ajouter au contenu de EAX. La directive `DWORD PTR` dans l'opérande source de la 2^e instruction indique la taille (4 octets) de la variable `i` vers laquelle pointe EBX.

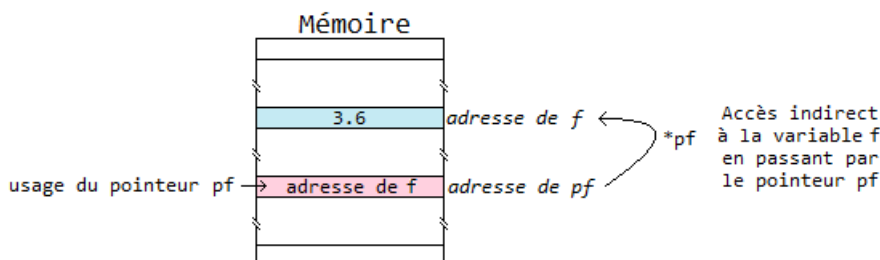
On voit clairement que **le pointeur n'est rien de plus qu'un intermédiaire pour accéder au contenu d'une variable en mémoire.**

Exemple: accéder indirectement à une variable du type float :

```
float f = 3.6, *pf = &f;
const float fc = 25;
```

Langage C	Assembleur
<pre>f = 25 + *pf;</pre>	<pre>movss xmm0, fc mov ebx, pf addss xmm0, DWORD PTR [ebx] movss f, xmm0</pre>

En C, quand on utilise `*pf`, on atteint la variable `f` en passant par le pointeur `pf` :



En assembleur, pour réaliser une opération similaire, on doit passer par un registre général ainsi :

```
mov ebx, pf          (1)
addss xmm0, DWORD PTR [ebx] (2)
```

L'instruction (1) copie dans EBX l'adresse de la variable f se trouvant dans le pointeur pf. L'instruction (2) passe par EBX utilisé comme pointeur pour récupérer la valeur de la variable f afin de l'ajouter au contenu de XMM0. La directive DWORD PTR dans l'opérande source de la 2^e instruction indique la taille (4 octets) de la variable f vers laquelle pointe EBX.

Les directives byte ptr, word ptr, dword ptr et qword ptr :

Dans l'opérande d'une instruction, on utilise une des directives suivantes en assembleur pour signaler la **taille** (et non pas le type) de l'emplacement en mémoire vers lequel on pointe.

<i>Directive en assembleur</i>	<i>Taille en mémoire</i>	<i>Type de données correspondant</i>
BYTE PTR	1 octet	char
WORD PTR	2 octets	short
DWORD PTR	4 octets	int/float/pointeur
QWORD PTR	8 octets	double

Exemple :

```
char b, *pb = &b;
short s, *ps = &s;
int i, *pi = &i;
float f, *pf = &f;
double d, *pd = &d;
```

<i>Langage C</i>	<i>Assembleur</i>
*pb = b + *pb;	movsx eax, b mov ebx, pb movsx ecx, BYTE PTR [ebx] add eax, ecx mov BYTE PTR [ebx], al
*ps = s + *ps;	movsx eax, s mov ebx, ps movsx ecx, WORD PTR [ebx] add eax, ecx mov WORD PTR [ebx], ax
*pi = i + *pi;	mov eax, i mov ebx, pi mov ecx, DWORD PTR [ebx] add eax, ecx mov DWORD PTR [ebx], eax
*pf = f + *pf;	movss xmm0, f mov ebx, pf movss xmm1, DWORD PTR [ebx] addss xmm0, xmm1 movss DWORD PTR [ebx], xmm0
*pd = d + *pd;	movsd xmm0, d mov ebx, pd movsd xmm1, QWORD PTR [ebx] addsd xmm0, xmm1 movsd QWORD PTR [ebx], xmm0

Pour une opération de lecture en mémoire, le pointeur est l'opérande source de l'instruction (exemple : `movsx ecx, byte ptr [ebx]`). Une fois que l'opération de lecture a été réalisée en passant par le pointeur, la donnée obtenue est traitée selon son type de données en appliquant les mêmes règles que celles vues dans les listes d'exercices précédentes.

Pour une opération d'écriture en mémoire, le pointeur est l'opérande destination de l'instruction (exemple : `movsd qword ptr [ebx], xmm0`).

Les opérateurs ++ et -- avec les pointeurs :

Exemple :

```
char  b, *pb = &b;
short s, *ps = &s;
int   i, *pi = &i;
float f, *pf = &f;
double d, *pd = &d;
const float ftmp1 = 1;
const double dtmp1 = 1;
```

<i>Langage C</i>	<i>Assembleur</i>
<code>(*pb)++;</code>	<code>mov eax, pb inc BYTE PTR [eax]</code>
<code>(*ps)++;</code>	<code>mov eax, ps inc WORD PTR [eax]</code>
<code>(*pi)++;</code>	<code>mov eax, pi inc DWORD PTR [eax]</code>
<code>(*pf)++;</code>	<code>mov eax, pf movss xmm0, DWORD PTR [eax] addss xmm0, ftmp1 movss DWORD PTR [eax], xmm0</code>
<code>(*pd)++;</code>	<code>mov eax, pd movsd xmm0, QWORD PTR [eax] addsd xmm0, dtmp1 movsd QWORD PTR [eax], xmm0</code>

Les castings :

Exemple :

```
short s, *ps = &s;
double d;
const double tmp = 2;
```

<i>Langage C</i>	<i>Assembleur</i>
<code>d = (double)*ps + 2;</code>	<code>mov eax, ps movsx ebx, WORD PTR [eax] cvtsi2sd xmm0, ebx addsd xmm0, tmp movsd d, xmm0</code>

La valeur codée sur 2 octets de la variable *s* qu'on récupère via le pointeur *ps* est étendue à 32 bits, puis elle convertie vers le type double et l'addition est alors réalisée sur des doubles.

Exemple général :

```
short s = -6, *ps = &s;
int i = 3, *pi = &i;
double d = 2.3, *pd = &d;
const double dTmp = 3.0;
```

Langage C	Assembleur
<p>The diagram illustrates the evaluation of the C expression <code>*pi = *pd * 3 + -(*ps);</code>. It shows the flow of data and operations, with numbered steps corresponding to the assembly code:</p> <ul style="list-style-type: none"> (1) <code>*pd</code> is converted to a pointer (1). (2) <code>3</code> is converted to a double (2). (3) <code>*ps</code> is converted to a pointer (3). (4) <code>-(*ps)</code> is converted to an integer (4). (5) The integer from (4) is converted to a double (5). (6) The double from (5) is converted to a double (6). (7) The double from (6) is added to the double from (2) (7). (8) The result from (7) is converted to an integer (8). (9) The integer from (8) is converted to a pointer (9) and stored in <code>*pi</code>. 	<pre> mov eax, pd // (1) movsd xmm0, QWORD PTR [eax] movsd xmm1, dTmp mulsd xmm0, xmm1 // (2) mov eax, ps // (3) movsx eax, WORD PTR [eax] // (4) neg eax // (5) cvtsi2sd xmm1, eax // (6) addsd xmm0, xmm1 // (7) cvtsd2si eax, xmm0 // (8) mov ebx, pi // (9) mov DWORD PTR [ebx], eax </pre>

Exercices

Écrivez la séquence d'instructions en assembleur qui correspond à chaque instruction d'affectation en langage C suivante ...

```
char b = -2, * pb = &b;
short s = 4, *ps = &s;
int i = -3, * pi = &i;
float f = 52.5, * pf = &f;
double d = 3.14, * pd = &d;
```

- `i = *pb + *pf;` // `i = 50`
- `*pi = -b + 2 * ~b | i + 0xfe;` // `i = 255`
- `*pd = *pf + (double)*pb + 0.2;` // `d = 50.700...`
- `s = ++s * -(*pi)-- / 2 * ~(*ps)--;` // `s = -43, i = -4`
- `*ps = -*pb * -(float)*ps * 3.5;` // `s = -28`
- `s = s * (*pi)-- / 2 + --b * (*ps)--;` // `s = -19, i = -4, b = -3`
- `i = ~i * (5.3 * (float)*pb);` // `i = -21`
- `*pd = (float)i * 2.45 + (int)*pf - ((int)(5.8 * b) | 0xff);` // `d = 45.649999...`
- `f = ++b + -(float)(*pd)-- * ++i;` // `f = 5.28000021, b = -1, d = 2.14, b = , i = -2`
- `*ps = -(int)((*pf)++ + (--b | 0xf) * (float)i) ^ 0xf00f * *pb;`
// `s = -12262, f = 53.5, b = -3`
- `*pi = (float)i * f + (int)*pf - 15435 / (float)*pb;` // `i = 7612`