```
|---------------------------|
| Programmation : POINTEURS |
|---------------------------|
```

Pointeurs

Un pointeur p vers un truc est un objet dont la valeur est l'adresse d'une
variable de type truc. Cet autre objet est note p^ en Pascal, *p en C.
p est alors de type "pointeur vers truc"

```
var i : integer ;   (* i est en 006 *)
var p : ^integer ;  (* p est en 003 *)
```

Dans le contexte qui suit, p pointe vers un entier qui vaut 9

```
| 001 |   9 |     (* 9 est un entier, c'est p^ (Pascal) ou *p (C) *)
| 002 |     |
| 003 | 001 |     (* 001 est une adresse, celle de *p, et c'est aussi *)
| 004 |     |                                        (* la valeur de p *)
| 005 |     |
| 006 |   7 |     (* 7 est un entier, c'est la valeur de i *)
```

writeln(p^ = 9) ; ecrira "true" a l'execution
writeln(p = 9) ; sera refuse par le compilateur Pascal : erreur de type
en C, vous aurez peut-etre un warning a propos des types, mais une adresse
etant codee par un entier, et C autorisant un peu tout et n'importe quoi,
il va laisser passer, et donnera faux car (001 <> 9)

Une facon imagee de comprendre les choses :
La difference entre un objet de type truc et un pointeur vers truc
est la meme que celle entre une maison (truc) et une carte de visite
qui donne l'adresse de la maison (pointeur vers truc)

```
| 001 | 005 | p
| 002 | 005 | q
| 003 | 006 | r
| 004 |     |
| 005 |   9 | *p *q
| 006 |   9 | *r
```

```
print(  p =  q    Vrai, 005  = 005
        p =  r    Faux, 005 <> 006
       *p = *q    Vrai, 9 = 9 (memes valeurs; normal c'est la meme variable)
       *p = *r    Vrai, 9 = 9 (ces deux variables ont la meme valeur)
      )
```

```
*r <- 8
print(*r)       "8"
q <- r
*q <- 7
print(*r)       "7"
```

```
| 001 | 005 | p
| 002 | 006 | q
| 003 | 006 | r
| 004 |     |
| 005 |   9 | *p
| 006 |   7 | *r *q
```

```
 p =  q   rend vrai ssi les cartes de visite portent les memes adresses
*p = *q   rend vrai ssi les maisons dont les adresses sont donnees par p et q
          sont jumelles
 p <-  q   je recopie sur la carte p l'adresse trouvee sur la carte de visite q
*p <- *q   je construit a l'adresse donnee par p, une maison identique a celle
          qui se trouve a l'adresse donnee par q
```

On peut avoir des pointeurs de pointeurs (frequents en C), i.e. des cartes qui
indiqueront ou se trouve la carte qui donne l'adresse de la maison.
On peut faire des pointeurs de pointeurs de pointeurs de pointeurs de...
(On pratique, on va rarement au dela du pointeur de pointeur)

```
int ***p

| 001 | 006 |   *p, pointeur de pointeur d'entier
| 002 |     |
| 003 | 001 |    p, pointeur de pointeur de pointeur d'entier
| 004 |     |
| 005 |   9 | ***p, entier
| 006 | 005 |  **p, pointeur d'entier
```

Pointeur NULL

Un pointeur est donc un objet dont la valeur est l'adresse de l'objet pointe.
On a besoin, en particulier avec les structures chainees, de pouvoir
dire qu'il n'y a pas d'adresse dans p, et donc de dire qu'il n'y a
pas d'objet pointe par p, ie que *p n'existe pas.
On veut en quelque sorte pouvoir avoir une carte de visite vierge,
toute blanche sans rien ecrit dessus.
Pour faire cela, on utilise un numero d'adresse, qui en fait ne
correspond a aucune adresse et code l'absence d'adresse : c'est nil
en Pascal, et NULL en C (ce code est en general 0 en Pascal,
toujours 0 en C).

```
var p : ^integer ;
int *p ;

| 001 | ... |  p

        p <- NULL ;

| 001 | 000  |  p

        writeln(p^) ;
```

Je demande a acceder a *p (pour lire son sa valeur et l'ecrire), qui est
la zone memoire dont l'adresse m'est fournie comme valeur de p.
C'est comme si j'etais macon, et que mon patron m'avait donne l'ordre
de reparer la maison dont l'adresse est sur la carte qu'il m'a
donnee, et qu'il m'a donne une carte de visite blanche.
Cette zone memoire n'existe tout simplement pas, mission impossible
qui se traduit par :
arret brutal du programme, Segmentation Fault, core dumped

malloc - new : allocation de memoire

(en Pascal : "new(p)" remplace "p <- ... malloc ...")

Quand on declare une variable de type pointeur mettons
vers un entier, l'espace memoire reserve est celui pour un pointeur
IE l'espace necessaire pour stoquer une adresse. Le compilateur ne
reserve pas de memoire pour un entier.

```
var p : ^integer ; (* le compilo reserve la memoire 003 *)

| 001 |     |     (* aucune memoire reservee qui corresponde a un entier *)
| 002 |     |
| 003 | 629 |     (* seule memoire reservee, dont le contenu est une adresse *)
| 004 |     |         (* comme on n'a pas initialise, on trouve une valeur  *)
| 005 |     |         (* parasite, par exemple 629, qui ne correspond a rien *)
| 006 |     |
```

Pour reserver de la memoire et la mettre au bout, on utilise new en Pascal,
malloc en C ;

new(p) : reserve de la memoire pour stoquer un objet du type de ce vers quoi p doit pointer, et ecrit l'adresse de cette memoire dans la memoire allouee pour p

```
var p : ^integer ;

| 001 |     |
| 002 |     |
| 003 | 629 |     (* memoire reservee, type ^integer *)

new(p) ;

| 001 | 752 |     (* memoire reservee, type integer *)
| 002 |     |
| 003 | 001 |     (* memoire reservee, type ^integer *)
```

En C, on utilise malloc : malloc prend en argument un entier x, reserve une zone memoire de x octets, et rend l'adresse de cette memoire, le resultat etant de type void*, ie pointeur compatible avec tous les pointeurs (ou encore "pointeur vers non-precise"). On obtient le nombre d'octets pour un type grace a sizeof. On ecrit donc :

```
        p = (int*) malloc(sizeof(int)) ;
```

int : le type entier ;
sizeof(int) : le nombre d'octet pour un entier, mettons 4 (si vous savez que votre compilateur prend 4 octets pour un int, utiliser tout de meme sizeof(int) plutot que 4. Sinon votre programme ne marchera plus sur d'autres machines)
malloc(sizeof(int)) : reserve 4 octets, rend l'adresse de ces 4 octets, cette adresse est une valeur de type pointeur void*
(int*) malloc(...) : le pointeur void* est convertit en pointeur int*, ce n'est qu'un changement de type. A la compilation, le compilo sera content car le type sera en accord avec celui de p. A l'execution, ca ne change rien
On a une expression dont la valeur est l'adresse de la zone memoire reservee
p = ... : et cette valeur de pointeur, qui est adresse de zone reservee, je l'ecris dans la variable pointeur p.

free - dispose : desallocation de memoire

free(p) en C, dispose(p) en Pascal

Le compilateur ne sait pas quand il peut recuperer une zone memoire allouee par un new/malloc. Il faut rendre a la memoire explicitement par un dispose/free

```
dispose(p) ; (* rend a la memoire la zone dont l'adresse est la valeur de p *)

| 001 | ???   |     (* cette memoire n'est plus reservee *)
| 002 |       |
| 003 | 001 ? |     (* memoire reservee, type ^integer *)
```

La ou il y des ???, je ne sais pas trop si le compilateur a fait qqch (ca va dependre du compilateur)

```
var p,q,r : ^integer ;
        new(p) ;
        new(q) ;
        p^ := 3 ;
        q^ := p^ + 2 ;

| 001 | 005 | p
| 002 | 006 | q
| 003 |     | r
| 004 |     |
| 005 |   3 | p^
| 006 |   5 | q^
```

```
        r := p ;
        r^ := q^ + 6 ;

| 001 | 005 | p
| 002 | 006 | q
| 003 | 005 | r
| 004 |     |
| 005 |  11 | p^ r^
| 006 |   5 | q^

        dispose(q) ;

| 001 | 005   | p
| 002 | 006 ? | q     (* suivant compilo, valeur(q) modifie ou non *)
| 003 | 005   | r
| 004 |       |
| 005 |  11   | p^ r^
| 006 | ????? |       (* cette zone rendue a la memoire *)

        dispose(p) ;

| 001 | 005 ? | p     (* suivant compilo, valeur(p) modifie ou non *)
| 002 | ..... | q
| 003 | 005   | r  (* par contre, pas de doute, r pointe toujours vers 005 *)
| 004 |       |
| 005 | ????? | (* zone rendue a la memoire, bien qu'elle reste accesible *)
| 006 | ..... |                                          (* via r !!!! *)
```

On remarque que r n'est pas modifie par dispose(p) mais que la variable r^
        est devenu quelquechose d'incongru : C'est la zone memoire qui
        etait reservee et qui ne l'est plus, et donc si j'utilise r^,
        je cours vers les gros problemes, avec plantage en regle a la clef.

 beurk !

Petit musee des horreurs, ou bestiaire de qu'il faut ecrire pour avoir
        0 a l'examen :

```
        new(p) ;  <----\
        p := q ;  <-------   HORREUR
```

Je reserve une zone memoire pour un entier dont l'adresse est conservee
        en p, puis j'ecrit quelquechose de neuf en p, qui donc ne pointe
        plus vers l'espace reserve par new. L'espace memoire reste reserve
        tant qu'il n'est pas libere par un dispose, lequel dispose est
        impossible a effectuer car on a perdu l'adresse de la zone memoire
        reservee par le new. Cette zone restera donc inutilement reservee
        tout le long du programme. C'est du gaspillage en bonne et due forme.
        N'allez pas pleurer si tout ca se termine par un "out of memory"

Note : ne pas confondre avec

```
        new(p) ;
        p^ := q^ ;
```

qui lui n'a rien d'horrible.

Meme principe pour :

```
        new(p) ;  <----\
        p := nil ;  <-------   HORREUR
                (* cf definition du nil ci-dessous *)

        new(tmp) ;              <--\
        tmp^ := 0 ;             <----\
        p := tmp ;              <------- HORREUR
        dispose(tmp) ;          <----/
        - utilisation de p^ -   <--/
```

Certes, on n'utilisera plus tmp (IE la carte de visite tmp), ce n'est pas une raison pour jeter la variable entiere pointee par tmp (le tableau regarde par tmp) qui reste utilisee et qui est accedee via p.

Exemple pratique (cf plus tard les listes chainees) :

```
procedure ajoute_en_tete ( x : integer ; var l : liste ) ;
begin
        new(tmp) ;
        tmp^.valeur := x ;
        tmp^.suite := l ;
        l := tmp ;
        dispose(tmp) ;   <-------- NON !!!! HORREUR
end ;
```

Ceux qui ecrivent les horreurs ci-dessus sont des gens qui croient - et ils ont tort a 100 % - que new(p) est une autorisation a se servir de p, et dispose(p) une indication que l'on a fini de se servir du pointeur p.

En Java : Il n'y a pas de dispose en Java, qui cherche les memoires perdues et elimine automatiquement. Pour cela, il maintient sur chaque variable le nombre de pointeurs qui lui pointe dessus. Quand ca atteint 0, Java fait un dispose. On notera que ca ne suffit ca, car il peut y avoir des "boucles", cf :

```
                new(p) ;
                p->suite := p ;
                p := nil
        ou encore
                new(p) ;
                new(q) ;
                p->suite := q ;
                q->suite := p ;
                p := nil ;
                q := nil ;
```

Pour detecter ces records qui sont perdus bien que pointes, Java est oblige d'utiliser un rammase-miette, ce qui est complique. On n'a pas de rammase miette qui ramasse tout rapidement sans avoir un cout important en temps.

```
var p,q,r : ^^integer ;
        new(r) ;
        new(r^) ;
        r^^ := 7 ;
        new(q) ;
        q^ := nil ;
        p := nil ;
        new(p^) ;
```

declaration de p,q,r : zone memoire reservee pour ces 3 variables

001		
002		
003		
004		
005		p type ^^integer
006		q type ^^integer
007		r type ^^integer

new(r) : reservation d'une zone memoire de type ^integer et stockage de l'adresse dans r :

001		r^ type ^integer
002		
003		
004		
005		p type ^^integer
006		q type ^^integer
007	001	r type ^^integer

new(r^) : reservation d'une zone memoire de type integer et stockage de
l'adresse dans r^ :

```
| 001 | 002  | r^  type  ^integer
| 002 |      | r^^ type   integer
| 003 |      |
| 004 |      |
| 005 |      | p   type ^^integer
| 006 |      | q   type ^^integer
| 007 | 001  | r   type ^^integer
```

r^^ := 7 : ecriture de 7 dans r^^

```
| 001 | 002  | r^  type  ^integer
| 002 |   7  | r^^ type   integer
| 003 |      |
| 004 |      |
| 005 |      | p   type ^^integer
| 006 |      | q   type ^^integer
| 007 | 001  | r   type ^^integer
```

new(q) : reservation d'une zone memoire de type ^integer et stockage de
l'adresse dans q :

```
| 001 | 002  | r^  type  ^integer
| 002 |   7  | r^^ type   integer
| 003 |      | q^  type  ^integer
| 004 |      |
| 005 |      | p   type ^^integer
| 006 | 003  | q   type ^^integer
| 007 | 001  | r   type ^^integer
```

q^ := nil ; ecriture de nil dans q^

```
| 001 | 002  | r^  type  ^integer
| 002 |   7  | r^^ type   integer
| 003 | 000  | q^  type  ^integer
| 004 |      |
| 005 |      | p   type ^^integer
| 006 | 003  | q   type ^^integer
| 007 | 001  | r   type ^^integer
```

p := nil : ecriture de nil dans p :

```
| 001 | 002  | r^  type  ^integer
| 002 |   7  | r^^ type   integer
| 003 | 000  | q^  type  ^integer
| 004 |      |
| 005 | 000  | p   type ^^integer
| 006 | 003  | q   type ^^integer
| 007 | 001  | r   type ^^integer
```

new(p^) :
reservation d'un espace memoire de type integer : ok
puis
stoquage de l'adresse de cet memoire dans p^ : mission impossible,
p vaut nil donc p^ n'existe pas :

=> Segmentation Fault

par contre, ce qui suit tourne ok :

```
new(p) ;
new(p^) ;
p^^ := 8 ;
dispose(p^) ;
dispose(p) ;
```

pointeurs non initialises

Une variable pointeur qui n'est pas initialise contient n'importe quoi,
et donc si je fais sans avoir initialise p :

```
print(*p) ;
*p <- 5 ;
```

Quand le vent viendra du sud, j'aurai un compilo qui initialise automatiquement
a nil, et donc l'acces a *p est incense, d'ou arret brutal, Seg. Flt
Quand le vent viendra du nord, j'aurai n'importe quoi dans p, et le hasard
fera que ce n'importe ne correspond pas a une adresse memoire.
L'acces a *p est incense : arret brutal, Seg Flt
Par temps calme, j'aurai n'importe quoi dans p, et le hasard
fera que ce n'importe correspond a une adresse memoire. Donc *p
existera mais ca sera n'importe ou, ce nimporte ou
pourra correspondre a une zone utilisee, ou bien a une zone
libre de la memoire. print(*p) affichera le contenu de ce
n'importe ou, d'ou affichage d'un entier n'importe quoi.
*p <- 5 ; est bien plus pervers, il ira ecrire 5 dans ce
n'importe ou, ce qui peut causer des degats pas immediatement
decelables mais qui conduiront le programme a sa perte
(ou celui du voisin si la gestion du systeme est mal securisee,
et vous laisse ecrire dans la memoire du voisin si vous en
avez l'adresse

Ne surtout pas confondre un pointeur qui vaut nil avec un pointeur pas
initialise.

Attention : quand on fait new(p), cela initialise p puisque cela ecrit
quelquechose de cense dans p. C'est p^ qui n'est pas initialise.

L'operateur & :

Quand on a une variable x (au sens large), &x designe en C l'adresse
de la variable x. Si x est de type truc, alors &x est (a pour valeur)
l'adresse d'un objet de type truc, et c'est donc de type
"pointeur vers truc", ce qui fait que l'on peut ecrire
(ce qui fait pointer p vers i) :

```
var p : ^integer ; (* p en 002 *)
var i : integer ;  (* i en 001 *)
        p := & i ;
| 001 |     | i
| 002 | 001 | p
| 003 |     |
```

Par contre, &x n'a pas d'adresse ie n'est pas une variable, tout comme 0
ou i+3 sont des entiers qui ne sont pas des variables.
En supposant que p^ existe (en particulier p <> nil), le test
" p = & (p^) " est correctement type et rendra vrai.
Il y a cependant une difference entre p et &(p^), c'est que
p est une variable, mais pas &(p^).
Si p est une variable de type pointeur vers entier, &p est parfaitement
legal, et c'est de type ^^integer (int** en C) ie de type
pointeur vers pointeur vers entier. On trouve
effectivement beaucoup de pointage double en C (passage par
adresse d'un pointeur, cf plus bas)
si exp ext une expression sans adresse, & exp n'a aucun sens,
en particulier : & 0 , &(n+1), &(&x) n'ont pas de sens,
par contre &(T[i]), &(structure.champs), &(*p) en ont.
On peut noter que *(&x) a un sens, c'est x, I mean, c'est la variable x,
ie meme adresse et donc meme valeur. L'expression *(&x) ne sert
cependant a rien, autant ecrire x directement

On a donc 4 facons d'initialiser un pointeur :

```
        p := nil ; je met ma carte de visite a blanc
        new(p) ;   J'achete un terrain sans vraiment decider lequel,
                   la maison dessus est comme elle peut,
                   je note l'adresse du terrain sur la carte de visite p
        p := q ;   Je copie l'adresse de la carte q sur la carte p
        p := & x ; Je marque sur p l'adresse de la maison x
```

beurk

```
        var p : ^integer ;
        var x : integer ;
                p := & x ;
                dispose(p) ; <---- HORREUR
```

Ce qui precede est une horreur car je tente de rendre a la memoire la zone
memoire de la variable x, qui n'a pas ete obtenue par un new/malloc
mais est une variable declaree.

```
procedure initialise_fantome (var p : ^integer) ;
var x : integer ;
begin
        p := & x ;  <---- OUH LA LA PAS BON
        x := 6 ;
end ;

        initialise_fantome(p) ;
        writeln(p^) ;
```

Du fait de l'initialisation, p a pour valeur l'adresse de x, mais cette
zone memoire x est geree par la procedure, qui en particulier
rend cette memoire a la fin de la procedure, donc initiliase_fantome
met en p l'adresse d'une zone memoire qui a ete allouee, mais ne
l'est plus. On peut y trouver nimporte quoi et le compilateur
peut en faire ce qu'il veut plus tard, c'est une variable
fantome. Donc le writeln
affiche n'importe quoi, c'est comme quand on faisait :

```
                new(q) ; q^ := 6 ; p := q ; dispose(q)
```

il y a meme garantie de plantage de programme si onutilise p^

```
procedure initialise_ok (var p : ^integer) ;
begin
        new(p) ;
        p^ := 6 ;
end ;

        initialise_ok(p) ;
        writeln(p^) ;
```

La tout va bien.