Pointeurs

Adresse et variables

Il existe en informatique un outil très puissant pour manipuler des données : il s'agit des pointeurs. Les pointeurs ont, à tort, une mauvaise réputation : "c'est difficile à employer, c'est incompréhensible, ça fait planter les programmes". Avec un minimum de rigueur, ils sont vraiment simples d'emploi et très souples !

Il suffit de se poser une simple question pour aborder et expliquer les pointeurs : mais où donc sont stockées les variables dans un ordinateur ? la réponse est quasiment immédiate : les variables sont stockées dans la mémoire vive (la RAM) de l'ordinateur, chaque variable est stockée à un endroit bien précis.

Nous ne connaissons rien de cet endroit, car lorsque nous définissons une variable, nous nous contentons de donner, son type et son nom. Comment l'ordinateur fait-il pour retrouver une variable dans la mémoire lorsqu'on lui demande de l'utiliser ?

En fait, dès qu'une variable est définie, elle a :

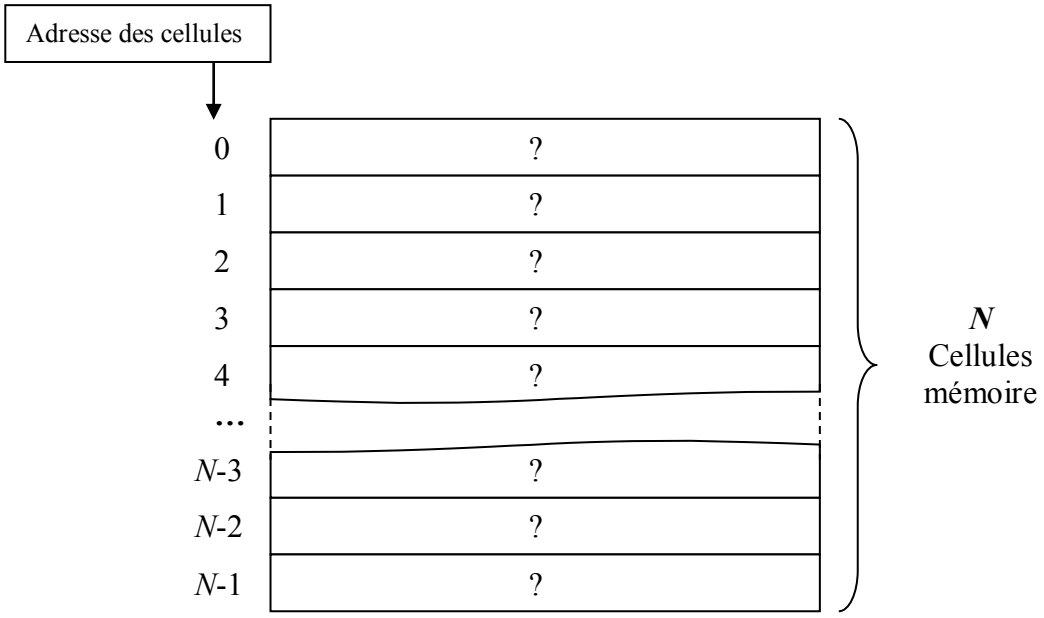
- un type, que nous choisissons;
- un nom, que nous choisissons aussi;
- une **adresse**, qui nous est inconnue car elle est gérée par l'ordinateur. cette adresse est l'endroit de la RAM où la valeur est stockée.

L'ordinateur, dès qu'il rencontre une définition de variable, fait une association entre le nom et l'adresse de la variable, et stocke cette association nom ⇔ adresse. Toutes les associations noms / adresses des variables sont stockées dans une table de correspondance. Donc, lorsque nous référençons (utilisons) une variable en indiquant son nom, l'ordinateur consulte cette table de correspondance et en déduit l'adresse de la variable, seule valeur exploitable pour lui.

C'est aussi grâce à cette table que l'ordinateur est capable de dire qu'une variable utilisée a été définie ou non, par exemple.

Représentation en mémoire

On peut symboliser la RAM d'un ordinateur comme un ensemble de cellules mémoire rangées de manière contiguë et repérées par leur adresse. Chacune de ces cellules contient en outre une valeur. Attention à bien faire cette distinction entre **adresse d'une cellule mémoire** et **valeur contenue dans la cellule mémoire**.



Chaque cellule contient une donnée de 8 bits (un octet). A un instant donné, une variable est stockée dans la RAM, donc dans une ou plusieurs cellules. Les points d'interrogation figurent les valeurs qui se trouvent dans les cellules, elles sont inconnues a priori.

Un exemple : soit une variable `a` définie comme un entier : `entier a;`

Lorsque l'on écrit cette définition, l'ordinateur attribue une adresse à la variable `a` (parmi les adresses disponibles) mais garde cette information pour lui, elle n'a en pratique que peu d'intérêt : qu'elle soit rangée en RAM à un certain endroit plutôt qu'à un autre n'a aucune influence sur cette variable.

Supposons que l'adresse attribuée à la variable `a` soit 1000, et que l'on écrive l'instruction : `a` ← `15;`. Alors, dans la mémoire, on trouvera :

1000	15

Que l'on peut lire : à l'adresse 1000 est stockée la valeur 15.

De plus, l'ordinateur stocke la correspondance : variable nommée `a` ⇔ adresse 1000.

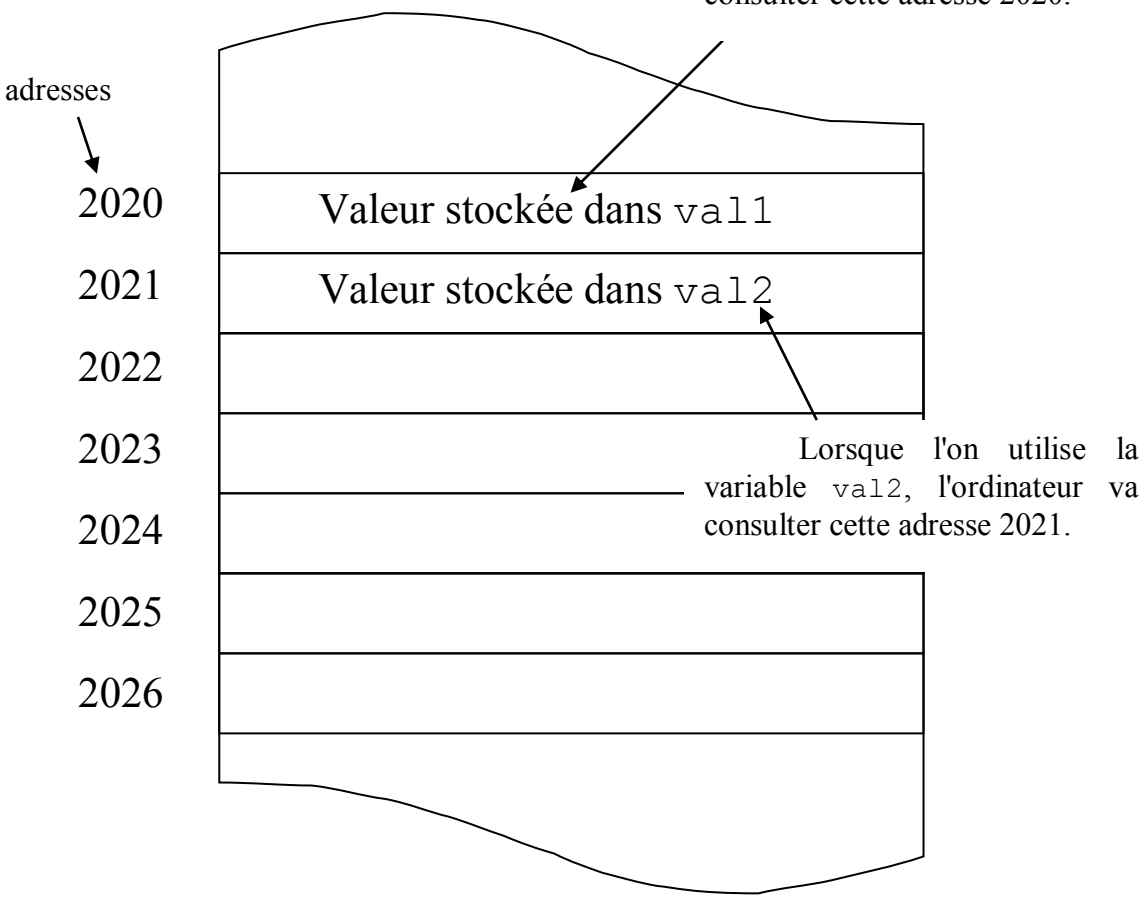
Prenons un programme simple comme exemple, pour matérialiser l'emploi de la table des correspondances entre noms de variable et adresses mémoires.

```
Programme exemple_adresses

entier val1,val2;

val1 ← 3;
val2 ← -2*val1+4;

si (val2 < val1)
{
     afficher(val2, " < ",val1);
}
```

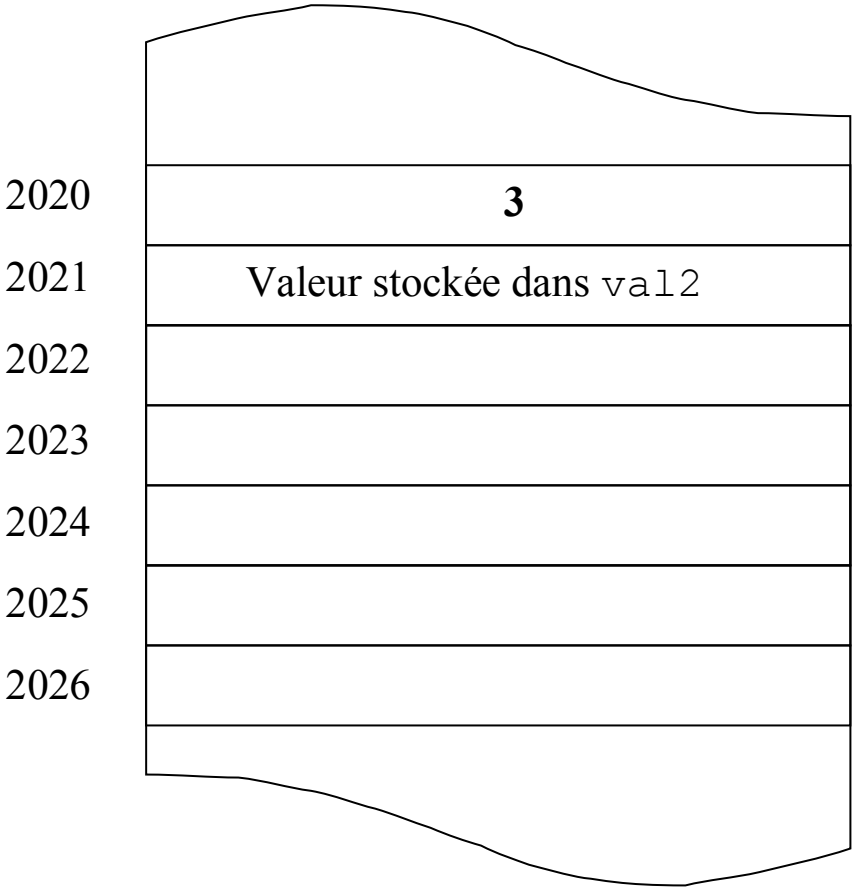
Lors des définitions de variables, l'ordinateur attribue des adresses aux variables `val1` et `val2`, supposons par exemple qu'il s'agisse des adresses 2020 et 2021 (ce qui, au fond, n'est pas fondamental).

`val1` ⇔ 2020

`val2` ⇔ 2021



effet de l'instruction : `val1 ← 3;` l'ordinateur consulte la table de correspondance, et stocke la valeur 3 à l'adresse 2020.



Effet de l'instruction : `val2 ← -2*val1+4;`. L'ordinateur calcule l'expression située à droite du symbole d'affectation : il accède à la variable `val1`, consulte la table de correspondance et récupère la valeur située à l'adresse 2020, il s'agit de la valeur 3. Il effectue le calcul, le résultat est –2, et cette valeur est rangée dans la variable `val2`. L'ordinateur consulte une nouvelle fois la table de correspondance et range la valeur –2 à l'adresse 2021.

La notation &

On peut accéder à l'adresse d'une variable en faisant précéder cette variable du caractère &.

Avec l'exemple précédent ; la variable a vaut 15, et l'adresse de a, notée &a, vaut 1000

La notation & suivie du nom d'une variable, indique l'adresse de la variable, qui est le numéro de la cellule mémoire où cette variable est stockée.

Le programme suivant, traduit en C et exécuté sur un ordinateur, donne une illustration de ce phénomène.

```
programme adresse
```

```
entier a;

a ←15;

afficher("valeur de a :",a);
afficher("adresse de a :",&a);
```

donne comme résultat :

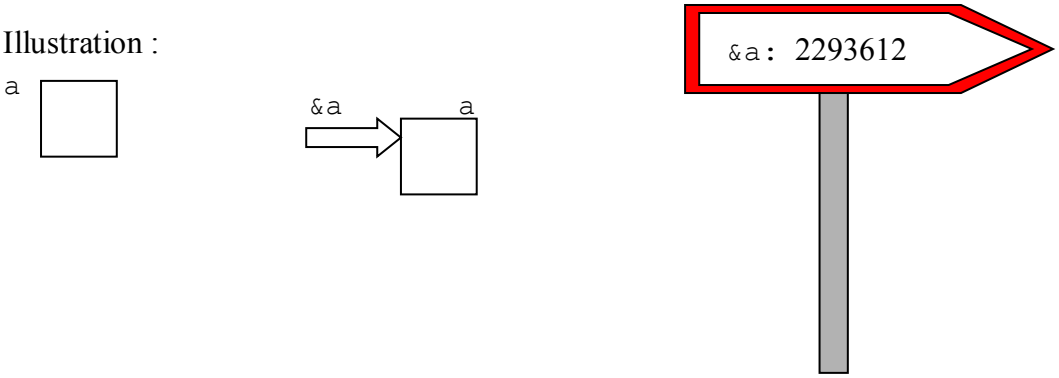
```
valeur de a : 15
adresse de a : 2293612
```

L'ordinateur a attribué une adresse à la variable `a`, là où il y avait de la mémoire disponible. Dans sa table de correspondance, on trouvera donc la relation :

`a` ⇔ 2293612

Si on représente la variable comme une boite contenant une valeur (ce que nous avons déjà fait dans le chapitre consacré aux variables et aux expressions), on représente l'adresse de cette variable (& devant le nom de la variable) comme un flèche ou un panneau indiquant où se trouve cette variable dans la mémoire.

Illustration :



La notation *

Il est possible de 'consulter' la mémoire, et d'obtenir une valeur, non en précisant un nom de variable, mais en précisant une adresse (un numéro de cellule mémoire). Il suffit de faire précéder cette adresse de la notation * pour récupérer la valeur contenue à cette adresse. Cependant, l'adresse ne peut pas dans ce cas être donnée par une valeur numérique, mais par une variable ou une expression. La valeur que l'on peut manipuler à partir de son adresse est appelée **contenu de l'adresse** ou **valeur pointée**.

& signifie : adresse de

*** signifie : contenu de**

types et occupation de la mémoire

Les types classiques (réel, entier, caractère) différent par les valeurs qu'ils peuvent stocker, mais aussi par le nombre d'octets (c'est à dire de cellules mémoires élémentaires) qu'une variable de chaque type occupe.

Avec un octet pour stocker une valeur, on se retrouve vite limité, car un octet étant un regroupement de 8 bits, on ne peut y stocker que 256 valeurs différentes !

Les types que nous utilisons permettent de stocker beaucoup plus de valeurs, tout simplement parce que la valeur d'une variable est stockée sur plusieurs octets. Le tableau ci-dessous récapitule le nombre d'octets occupé par une variable, selon son type

- Un entier occupe : 4 octets;
- Un réel occupe : 8 octets;
- Un caractère occupe : 1 octet.

Ainsi, si l'on définit 2 variables réelles dans un programme, la première sera stockée à une adresse *N*, la deuxième à l'adresse *N*+8 (et non *N*+1).

Les pointeurs

Qu'est-ce qu'un pointeur ?

Un pointeur est une variable dont la valeur est une adresse : cela correspond à la flèche ou au panneau représenté dans l'exemple précédent.

Comme pour toute variable, on peut la représenter par une boîte, mais celle-ci ne contient ni un entier, ou une valeur à virgule, ou encore un caractère. Elle contient l'adresse en mémoire d'une valeur. Pour la symboliser, on emploie donc une boîte de laquelle sort une flèche indiquant la position en mémoire qui est désignée. Nous allons abondamment illustrer ceci en donnant de nombreux exemples par la suite.

Type pointé

Lorsque l'on définit un pointeur, on doit indiquer quel est le type de la valeur qui se trouve à l'adresse stockée dans le pointeur.

Une adresse seule ne suffit pas à pouvoir récupérer une valeur, car nous avons vu précédemment que les différents types n'étaient pas tous stockés sur le même nombre d'octets. Ainsi, si l'on dispose d'une adresse, par exemple l'adresse 10 000, on ne peut pas récupérer ce qui s'y trouve si on ne sait pas s'il s'agit d'un caractère, d'un entier, ou d'un réel.

Dans le premier cas, il faut récupérer un seul octet, celui stocké à l'adresse 10 000,

Dans le deuxième cas, il faut récupérer 4 octets, stockés de l'adresse 10 000 à l'adresse 10 003,

Dans le cas d'un réel, il faut récupérer 8 octets, stockés de l'adresse 10 000 à l'adresse 10 007.

Cela, l'ordinateur n'est pas capable de le deviner à partir de l'adresse seule !

Lorsque l'on définit un pointeur, on doit indiquer si celui-ci permet de traiter des caractères, des entiers ou des réels. (nous verrons plus tard que l'on peut traiter des types composés définis par le programmeur, et la démarche sera la même que pour les types simples en ce qui concerne les pointeurs).

définition de variable

un pointeur, comme toute variable, est définie par un nom, puis, comme nous venons de le voir, par le type de la valeur qu'il permet de traiter (rappel : que l'on appelle aussi **valeur pointée** ou **contenu du pointeur**). La syntaxe de définition d'un pointeur est la suivante :

```
type_pointé *nom_du_pointeur;
```

dans ce cas, on dira que le pointeur pointe **vers** ou **sur** le type_pointé.

exemples :

On veut utiliser un pointeur nommé `p_ent`, qui permettra de pointer vers un entier. Cela signifie que lorsque l'on voudra accéder à la valeur située à l'adresse donnée par le pointeur, on accèdera à un entier. Ce pointeur est alors défini de la manière suivante :

```
entier *p_ent;
```

Cette définition se lit : `p_ent` est un pointeur qui pointe vers un entier.

Il ne faut pas confondre avec la définition d'un entier : `p_ent` n'est pas un entier ! la valeur de `p_ent` est bien une adresse. C'est à cette adresse en mémoire que l'on trouvera un entier.

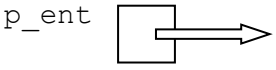
On peut aussi définir des pointeurs sur d'autres types. Les définitions suivantes sont tout à fait possibles :

```
reel *p_reel;
caractere *p_carac, *ad_car;
```

Par convention (ce n'est pas une obligation, puisque les noms de pointeurs suivent les mêmes règles de nommage que les variables, puisque, rappelons-le, les pointeurs sont des variables), on choisit de commencer le nom d'un pointeur par `p_` .

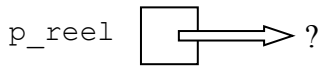
représentation d'un pointeur : on représente un pointeur par une boîte (comme pour les autres variables) de laquelle est issue une flèche indiquant que le pointeur désigne une adresse en mémoire.

Exemple :



que vaut un pointeur ?

Lorsqu'il est défini, un pointeur est comme toute autre variable : il contient une valeur inconnue et inexploitable, il ne faut donc pas l'utiliser. On matérialise ce fait en le représentant de la manière suivante (on utilise le pointeur `p_reel` de l'exemple précédent).

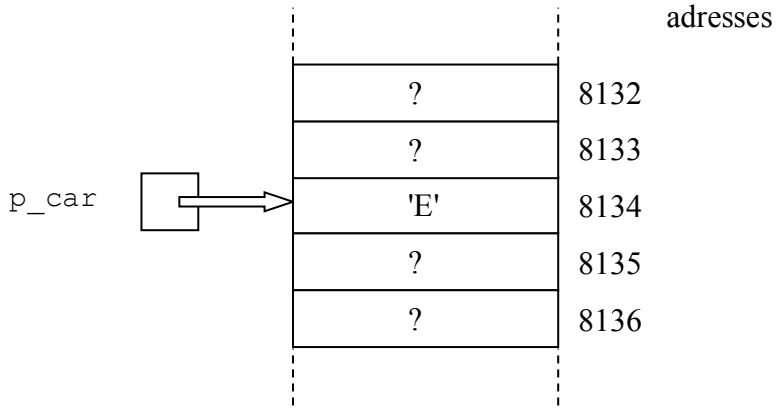


notation * et manipulation de la valeur pointée

Soit une variable nommé `p_car`, qui est un pointeur sur un caractère (`p_car` aura donc été défini par : `caractere *p_car;` On suppose que ce pointeur a été initialisé et que sa valeur est donc l'adresse en mémoire où se trouve un caractère.

Pour illustrer ceci avec un exemple, on suppose que le caractère est stocké à l'adresse 8134 et qu'il vaut 'E'. (mais cela est arbitraire, on aurait pu choisir d'autres valeurs).

Dans ce cas, on peut indiquer que le pointeur `p_car` pointe vers une cellule mémoire d'adresse 8134 et contenant le caractère 'E' en employant la représentation suivante :



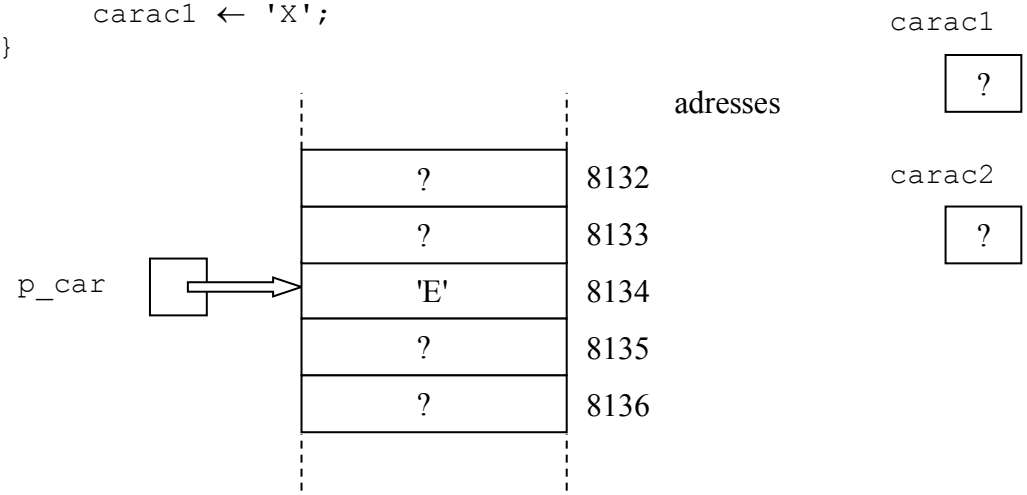
Le pointeur `p_car` a donc la valeur 8134 (c'est une adresse), mais on voudrait accéder à la valeur du caractère. Cette valeur est la valeur pointée par `p_car` (ou encore le contenu de `p_car`), on y accède donc par la notation *.

`*p_car` est la valeur pointée par `p_car`, et vaut 'E'. En fait, on peut tout à fait manipuler `*p_car` comme n'importe quel autre caractère.

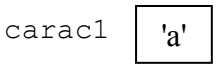
Il est donc possible d'écrire les instructions suivantes, dont nous allons voir les effets :

On suppose que l'on a défini d'autres variables de type caractère par la définition multiple : `caractere carac1, carac2;`

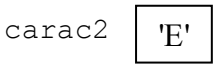
```
carac1 ← 'a';
carac2 ← *p_car;
*p_car ← carac1 + 1;
*p_car ← *p_car+1;
 si (carac2 < *p_car) alors
{
     carac1 ← 'X';
}
```



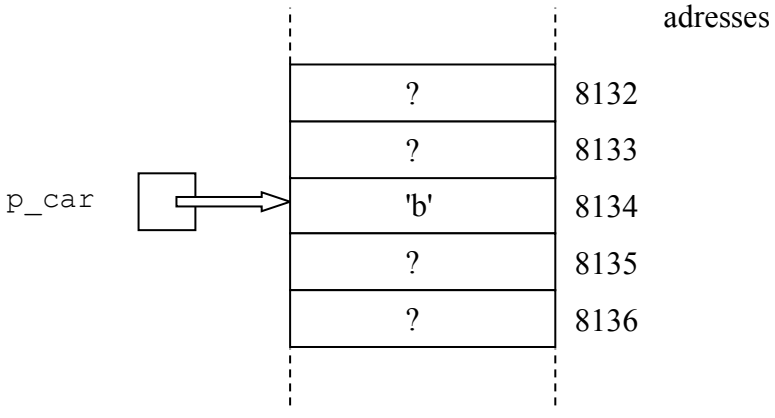
Instruction `carac1 ← 'a';` : c'est une affectation classique



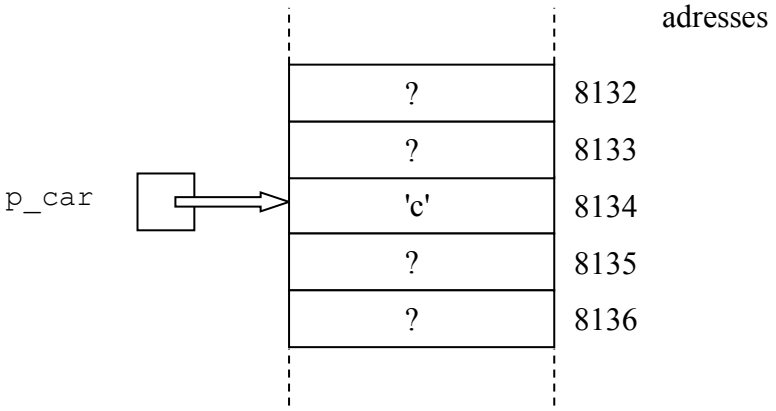
Instruction `carac2 ← *p_car;` : `*p_car` est le contenu de `p_car`, donc vaut 'E', on affecte cette valeur 'E' à la variable `carac2`.



Instruction `*p_car ← carac1+1;` : l'ordinateur calcule `carac1+1`, qui vaut 'b'. Cette valeur 'b' est affectée à `*p_car`, donc en mémoire à l'adresse pointée par `p_car`.



Instruction `*p_car ← *p_car+1;` : l'ordinateur calcule `*p_car + 1`, qui vaut 'c' (en effet, `*p_car` vaut 'b' à cause de l'instruction précédents). Cette valeur 'c' est affectée à `*p_car`, donc en mémoire à l'adresse pointée par `p_car`



En cas de doute sur l'emploi de cette notation *, qui est le plus souvent la source de la confusion pour l'emploi des pointeurs, il faut une fois de plus se baser sur les types des expressions écrites, cela permet de lever quasiment toutes les ambiguïtés.

Ainsi dans l'exemple précédent : `p_car` n'est pas de type caractère (c'est un pointeur vers un caractère), donc on ne peut pas le manipuler comme un caractère. `*p_car` est un caractère puisqu'il s'agit du contenu de (ou de la valeur pointée par) `p_car`. On peut donc le manipuler comme un caractère.

Retour sur la syntaxe de la définition d'un pointeur

Nous avons utilisé la notation * à deux occasions : pour définir un pointeur d'une part, et pour accéder à son contenu d'autre part. Cela pourrait être gênant, mais en réalité, cette notation est cohérente. Pour éviter toute confusion à l'avenir sur la signification de cette notation *, on peut présenter la définition d'un pointeur de manière un peu différente, mais qui ne change pas ce que nous en avons déjà dit précédemment.

Lorsque nous définissons un pointeur (par exemple vers un réel), nous écrivons :

```
reel *pointeur;
```

Or, * signifie contenu. Nous pouvons également lire cette définition sous la forme :

`*pointeur` est de type réel, ce qui signifie : le contenu de `pointeur` est de type réel, ou encore : la valeur pointée par `pointeur` est de type réel, ce qui est effectivement le cas.

Ainsi, une définition de pointeur se fait en indiquant le type du contenu du pointeur, ce qui permet de dire que la notation * signifie tout le temps : **contenu de** ou **valeur pointée par**.

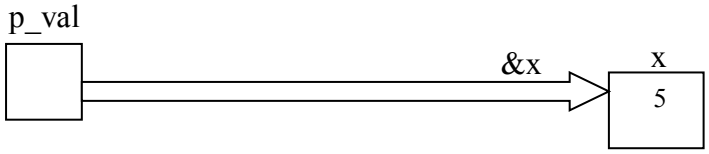
pointer vers une variable existante

Un pointeur peut contenir l'adresse d'une variable existante dans le programme (puisque toute variable définie dans un programme possède une adresse). Il suffit simplement que le type de la variable et le type pointé par le pointeur soient les mêmes.

Exemple :

```
Programme pointe_sur_var
reel x;
reel *p_val;

x ← 5;
```


```
p_val ← &x;
afficher("p_val vaut ", p_val ," et *p_val vaut :", *p_val);
```



Il s'agit du moyen le plus simple d'initialiser un pointeur pour qu'il contienne une adresse valide, à laquelle on peut accéder avec l'opérateur * de prise de contenu.

test de validité sur la valeur NULL

Contrairement aux autres types déjà rencontrés, il est possible de savoir si un pointeur est exploitable ou non (c'est à dire qu'il a déjà été initialisé et que l'on peut manipuler son contenu). En effet, parmi toutes les valeurs que peut prendre un pointeur, c'est à dire parmi toutes les adresses de la RAM, certaines ne correspondent pas à des zones exploitables. Cet état de fait est du au système d'exploitation, dont on sait qu'il occupe certaines portions de la mémoire pour fonctionner. Pour s'en rendre compte, il suffit de consulter les adresses des variables créées par un programme en langage C quelconque : ce sont des valeurs relativement élevées, ce qui signifie que toute la mémoire n'est pas a priori destinée aux variables de nos programmes en C (l'ordinateur a beaucoup d'autres choses à faire et occupe une bonne partie de la mémoire). Entre autres, l'adresse 0, nommée NULL en informatique, est une adresse que le système utilise pour stocker des informations qui lui sont vitales : on est certain de ne jamais pouvoir y trouver la valeur d'une variable d'un programme quelconque.

Ainsi, lorsque l'on définit un pointeur, on peut l'initialiser avec la valeur NULL, ce qui signifie que l'on ne peut pas encore l'exploiter. Cette convention d'utilisation est abondamment utilisée par le langage C ou d'autres langages.

```
programme test_pointeurs

caractere lettre;
caractere *p_c1, *p_c2;

lettre ← 'E';
p_c1 ← NULL;      // p_c1 vaut NULL, non accessible
p_c2 ← &lettre;   // p_c2 pointe vers la variable lettre

// testons si on peut acceder à *p_c2 (p_c2 initialisé)

si (p_c2 ≠ NULL) alors
{
     afficher("il est possible d'accéder a *p_c2 : *p_c2 vaut ",
*p_c2);
}

// testons l'initialisation de p_c1

si (p_c1 = NULL) alors
{
     afficher("p_c1 non initialise, *p_c1 non accessible");
}
```

en résumé

Pour être certain qu'un pointeur stocke une adresse qui est valide : on peut l'initialiser avec l'adresse d'un variable existante. S'il n'est pas possible de l'initialiser tout de suite, on peut lui affecter la valeur NULL.

Pour tester si on a le droit d'accéder au contenu du pointeur avec la notation *, on peut faire un test sur l'adresse stockée dans le pointeur. Si cette adresse est NULL, alors il ne faut pas utiliser la notation *, car cela provoque un plantage du programme.

l'allocation dynamique de mémoire

Pour exploiter totalement les pointeurs, il est indispensable de traiter l'allocation dynamique. Pour cela, retournons un peu en arrière pour revenir sur les tableaux. Ceux-ci ont le défaut d'avoir une taille maximum fixée lors de l'écriture du programme : certaines variables sont inutilisées, ce qui peut conduire, pour des applications utilisant beaucoup de tableaux, à une mauvaise gestion de la mémoire (beaucoup d'espace inutilisé).

Grâce aux pointeurs, on peut optimiser cette gestion de mémoire en obtenant l'espace nécessaire au stockage d'un nombre donné de variables; mais a la différence des tableaux, ce

nombre ne doit pas obligatoirement être connu lors de l'écriture du programme, mais est variable.

Le système d'exploitation est responsable de la gestion de la mémoire, il garde en permanence une carte de la mémoire utilisée et de la mémoire disponible. Lorsque l'on désire stocker ou utiliser plusieurs variables ou valeurs du même type, il suffit de demander au système d'exploitation s'il existe dans la mémoire disponible assez de place pour stocker ses valeurs : dans le cas où cela est possible, le système d'exploitation indiquera **l'adresse** en mémoire à laquelle se trouve une zone disponible pour stocker le nombre de valeurs demandées.

Comment réserver de la mémoire ?

Il suffit de le demander poliment au système d'exploitation par l'intermédiaire de la commande `reservation()`, à laquelle il faut fournir le nombre de variables que l'on veut stocker dans la mémoire, ainsi que leur type, puisque chaque type occupe plus ou moins de mémoire (un `reel` occupe plus de place qu'un `entier`, qui occupe lui-même plus de place qu'un `caractere`).

Syntaxe : `reservation((expression entiere) type des valeurs à stocker);`

L'expression entière (qui peut donc être une constante, une variable ou un calcul dont le résultat est de type entier) indique le nombre de variables que l'on veut stocker (ce qui est analogue à la taille utile d'un tableau). Le type est l'un des types de base connus.

Exemples d'utilisation :

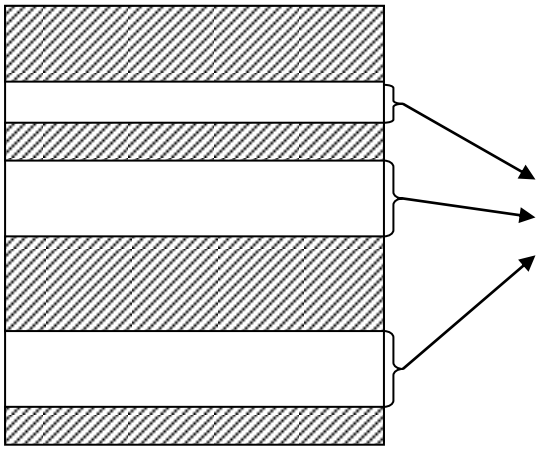
```
reservation(5 entier); // indique l'adresse d'une zone de mémoire où on peut stocker jusqu'à 5 valeurs de type caractère

reservation(1000 caractere); // idem, mais on pourra stocker jusqu'à 1000 caractères

reservation(n entier); // indique l'adresse d'une zone où on peut stocker n entiers, n étant une variable entière.

reservation(3*m+1 reel); // indique l'adresse d'une zone où on peut stocker 3*m+1 reels, m étant une variable entière.
```

Cette commande peut être utilisée lors du déroulement d'un programme, c'est pour cela que l'on parle d'allocation (de mémoire) dynamique.

Stockage de l'adresse

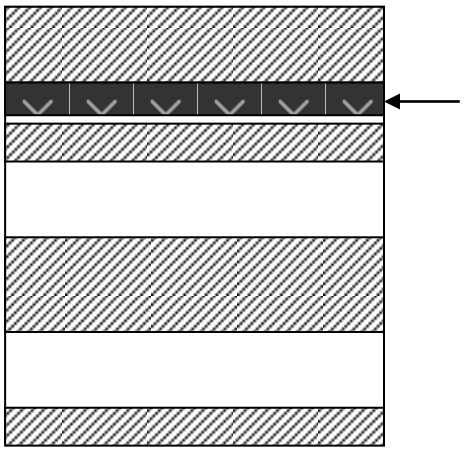
Il faut cependant stocker l'adresse que donne le système d'exploitation, car la zone de mémoire obtenue est alors considérée comme étant indisponible.

Illustration avec schéma de la mémoire : les zones hachurées représentent les parties de la mémoire déjà utilisées par le système d'exploitation ou par d'autres programmes en cours d'exécution.



On souhaite, par exemple, obtenir une zone mémoire permettant de stocker 30 valeurs de type `reel` : ceci s'obtient par la commande : `reservation(30 reel);`. On suppose que, sur le schéma de la mémoire, les zones libres permettent de stocker, respectivement de haut en bas , 40 `reel`, 100 `reel`, 300 `reel`.

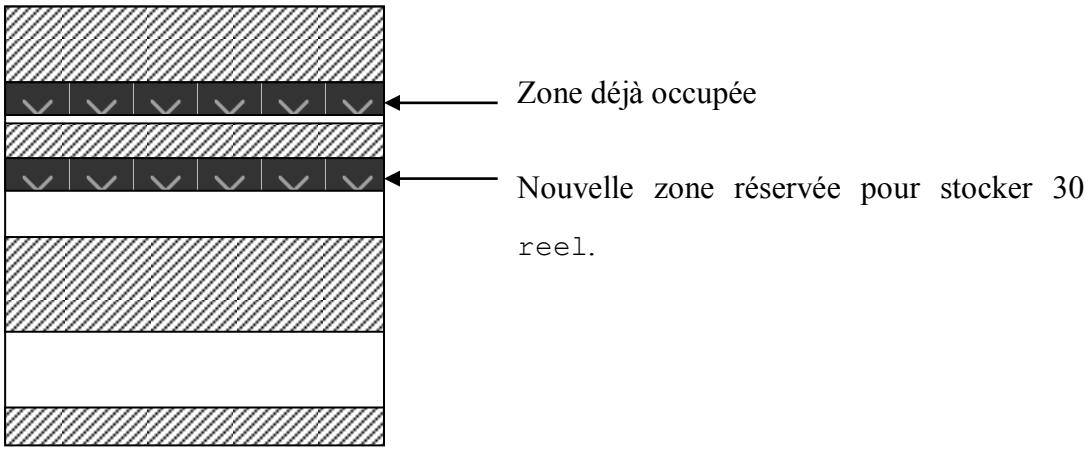
Si l'on se contente d'écrire cette instruction, voici ce que le système va comprendre :



A partir de cet instant, cette zone repérée par le motif  est considérée comme occupée, puisqu'elle a été réservée.

Cependant, comment peut-on accéder à et manipuler cette zone ?

On pourrait penser qu'il suffit de ré écrire l'instruction `reservation(30 reel);`. Cependant, dans ce cas, le système va comprendre que l'on demande une nouvelle zone, auquel cas l'effet de l'instruction va être le suivant :



Chaque instruction `reservation()` sera considérée comme une nouvelle demande par le système, et fournira donc une adresse différente.

Il est donc absolument indispensable de stocker l'adresse donnée par l'instruction réservation dans une variable. Or une variable qui stocke une adresse est, par définition, un pointeur. On doit donc systématiquement utiliser l'instruction `reservation()` avec la syntaxe suivante :

pointeur ← reservation(n type);

exemple avec une zone contenant des reels :

```
programme alloue_zone
reel *p_reel;
p_reel ← reservation(30 reel);
```

Echec d'une réservation de mémoire :

L'exécution de la réservation n'est pas obligatoirement couronnée de succès ! Il se peut que le système ne trouve pas de zone mémoire correspondant à la demande faite, pour diverses raisons. Même si en pratique ce cas ne se présente que très rarement, il faut prévoir un traitement adapté. Lorsque le système d'exploitation ne trouve pas de zone correspondante,

il indique NULL comme adresse. Ainsi, si le pointeur auquel on affecte l'adresse donnée par la réservation vaut NULL, c'est que la réservation n'a pas été effectuée.

Libérer la mémoire obtenue par `reservation`

Toute demande faite de réservation faite au système entraîne que le système considère la zone récupérée comme occupée. Cela évite qu'un autre programme ne vienne manipuler la zone de mémoire, ce qui pourrait s'avérer catastrophique. Il appartient donc au programmeur de restituer ou libérer cette zone de mémoire, soit dès qu'il estime ne plus en avoir besoin, soit à la fin du programme de toute manière. Pour effectuer cette libération, il suffit d'utiliser la commande :

```
liberer(pointeur);
```

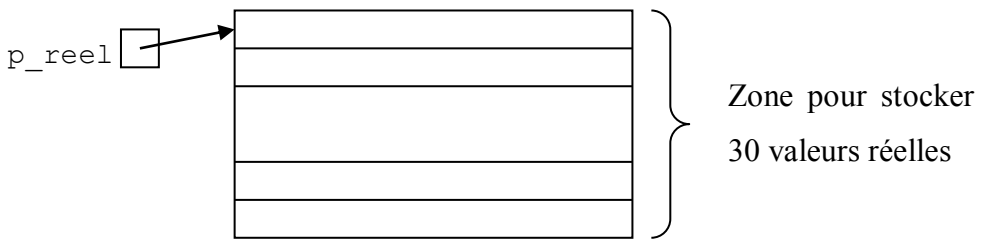
Où pointeur est un pointeur (quelle surprise….) stockant l'adresse de la zone de mémoire à libérer. Cette zone doit être une zone obtenue par la commande `reservation`, seulement dans le cas où celle-ci a réussi. Si une zone de mémoire obtenue par la commande `reservation` n'est pas libérée à la fin du programme, elle sera toujours considérée comme occupée par le système, et il n'y aura plus aucun moyen d'y accéder !

A toute réservation réussie doit correspondre une libération de mémoire !

Accéder aux valeurs contenues dans une zone de mémoire

Une fois la zone de mémoire allouée, comment faire en sorte de pouvoir accéder à n'importe quelle valeur stockée dans cette zone ? L'adresse stockée dans le pointeur indique simplement le début de cette zone. Continuons l'un des exemples d'allocation vus précédemment :

```
programme alloue_zone

reel *p_reel;

p_reel ← reservation(30 reel);
```



On voudrait, comme pour les tableaux, accéder à l'une de ces 30 valeurs de manière simple. Il existe pour cela deux méthodes, que l'on peut employer indifféremment, elles ont toutes deux leurs avantages et leurs inconvénients.

A partir de l'adresse stockée dans le pointeur

La notation * permet de manipuler une zone de mémoire obtenue dynamiquement par la commande `reservation`. Pour cela, nous devons nous intéresser quelque peu à l'arithmétique des pointeurs. Il est en effet possible d'effectuer des opérations arithmétiques simples comme l'addition et la soustraction avec des pointeurs et des entiers, et nous allons en détailler les différents effets.

Additionner un pointeur et un entier :

Soit `p` un pointeur vers un type simple (`caractere`, `entier` ou `reel`), et `d` une expression entière (constante, variable, ou calcul, au même titre qu'un indice de tableau).

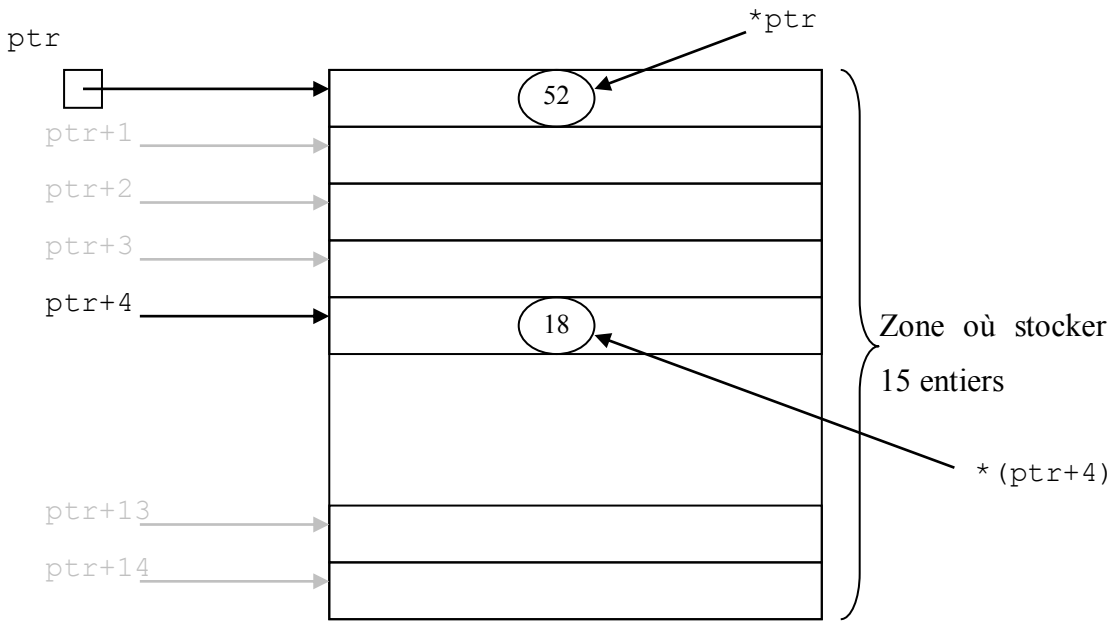
On peut alors utiliser l'écriture `p+d` ou `p-d`, même si les deux termes de l'opération ne sont pas du même type. Effectuer l'une de ces opérations revient en fait à calculer une nouvelle adresse (donc le résultat d'une de ces opérations est de type pointeur), et l'ordinateur utiliser les règles suivantes pour effectuer le calcul.

`p+d` donne l'adresse de la valeur située en $d^{ième}$ position dans la zone de mémoire considérée. L'expression entière `d` joue dans ce cas le rôle d'un indice de tableau. en appliquant l'opérateur `*` à cette adresse `p+d`, on peut obtenir la valeur stockée à l'adresse `p+d`.

Prenons un exemple pour illustrer ce calcul :

```
programme arith_ptr

entier a;        // a stocke un entier
entier *ptr;     // ptr stocke une adresse

ptr ← reservation(15 entier); // ptr stocke l'adresse d'une zone où
                              // 15 entiers peuvent être stockés

a ← 18;

*ptr ← 52;       // le contenu de ptr reçoit la valeur 52
*(ptr+4) ← a;    // la valeur 18 est rangée en mémoire à l'adresse
                 // ptr+4

afficher("a l'adresse ", ptr+4, " se trouve la valeur
",*(ptr+4),"\n");
```

schéma correspondant :



Il faut porter ici une attention particulière à l'emploi des parenthèses, car l'opérateur de contenu `*` est prioritaire sur l'opérateur d'addition `+`.

La ligne suivante :

```
afficher("*ptr+4 vaut : ", *ptr+4," et *(ptr+4) vaut : ", *(ptr+4));
```

affichera :

```
*ptr+4 vaut 56 et *(ptr+4) vaut 18
```

car `*ptr+4` est calculé comme : 1) récupération de `*ptr`, qui vaut 52 puis 2) addition de 4 : 56

et `*(ptr+4)` est calculé comme 1) calcul de l'adresse `ptr+4` puis 2) récupération de ce qui est stocké à cette adresse, soit 18.

Soustraire un entier à un pointeur permet d'obtenir une adresse située avant ce pointeur dans la mémoire. On ne se sert que très rarement de la soustraction, mais beaucoup de l'addition, car un pointeur désigne dans la majorité des cas l'adresse de début d'une zone de mémoire, et non celle de la fin de cette zone.

Avec un autre pointeur

Autre technique pour accéder à une valeur située dans la zone mémoire allouée : utiliser un autre pointeur que celui qui désigne cette zone. Ce second pointeur peut être modifié à loisir.

Prenons un exemple d'un programme qui alloue une zone de mémoire pour stocker 10 valeurs de type `reel`, qui initialise toutes les valeurs de cette zone, et qui doit afficher toutes les valeurs (un peu comme avec un tableau).

```
programme affiche_vals_zone

entier compt;
reel *ptr, *qtr; // deux pointeurs

ptr ← reservation(10 reel); // ptr reçoit l'adresse de la zone en
                            // question

qtr ← ptr; // qtr contient la même adresse que ptr : les deux
           // pointeurs désignent le même endroit de la mémoire

// ce commentaire remplace les instructions nécessaires à
// l'initialisation des valeurs stockées dans la zone mémoire

// affichage des valeurs en utilisant le pointeur qtr

pour compt de 0 à 9
{
     afficher(" la valeur d'indice ",compt," vaut ", *qtr,"\n");
     qtr ← qtr+1;
}
```

l'instruction `qtr ← qtr+1` permet de faire pointer `qtr` successivement sur toutes les valeurs de la zone de mémoire allouée par `reservation()`.

Dans les deux cas, il faut également faire attention à toujours conserver la valeur de l'adresse du début de la zone. Si cette valeur est modifiée, il n'existe alors aucun moyen de la récupérer !

Exemple d'une mauvaise manipulation :

```
programme parcours_zone

caractere *p_car;
entier cpt;

p_car ← reservation(20 caractere);

// on suppose que les valeurs de la zone sont initialisées

pour cpt de 0 à 20 faire
{
     afficher(*p_car);
     p_car ← p_car+1; // le pointeur p_car est modifié. Dangereux !
}
```

```
afficher("\n");
```

Le problème est que maintenant, plus aucune variable ne stocke l'adresse de début de la zone allouée, on ne peut donc plus y accéder…

la dualité tableaux ⇔ pointeurs

En réalité, un tableau est un pointeur ! Les différentes variables stockées dans un tableau se situent en effet à des positions ou adresses consécutives en mémoire : il s'agit donc du même type de zone de mémoire que celle que l'on obtiendrait en utilisant une allocation dynamique.

Un tableau est en réalité un pointeur : il stocke l'adresse à laquelle se situe la zone de mémoire

Les types :

`tableau de type (type tab[N])` et `pointeur vers type (type *)` sont analogues, à ceci près qu'un tableau est considéré comme une variable dont la valeur ne peut pas être modifiée : il stocke toujours la même adresse. Si l'on tente d'affecter une nouvelle adresse à un tableau, le compilateur indiquera un message d'erreur.

illustration :

```
programme tab_point

entier *ptr;
entier tab[12];

ptr ← reservation(25 entier);   // autorise
tab ← ptr;                      // interdit : tab n'est pas
                                // modifiable
ptr ← tab;                      // autorise
tab ← reservation(10 entiers);  // interdit : tab n'est pas
                                // modifiable
```

Lorsque l'on écrit l'instruction `ptr ← tab`, cela signifie que `ptr` va stocker l'adresse de la zone mémoire dans laquelle se trouvent les valeurs des variables du tableau. `ptr` et `tab` ont alors la même valeur (finalement, ici, on ne fait que recopier la valeur d'une variable dans une autre variable).

Cette équivalence va même plus loin, comme nous allons le voir dans le paragraphe suivant.

*Equivalence des notations [] et **

Nous venons de voir que les pointeurs ainsi que les tableaux stockent des adresses auxquelles on peut trouver certaines valeurs, qui sont rangées de manière consécutive dans la

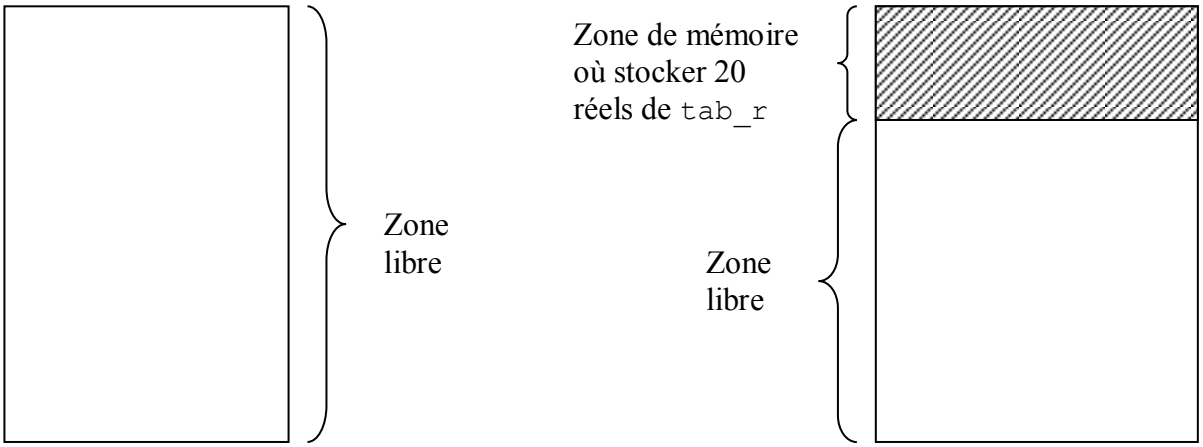
mémoire de l'ordinateur. Les notations : [] utilisée avec les tableaux, et * utilisée avec les pointeurs sont en réalité équivalentes !

Puisqu'un tableau est un pointeur, cela signifie qu'il stocke une adresse : cette adresse est celle de la zone de mémoire où sont rangées les variables du tableau.

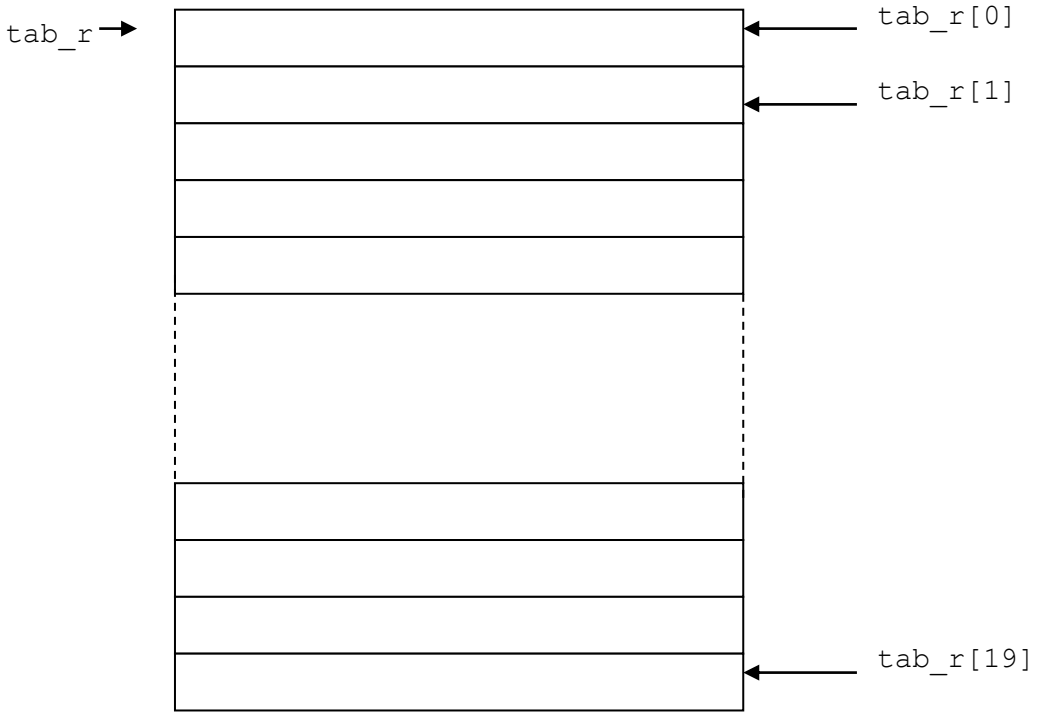
Illustration avec un tableau de reel :

```
reel tab_r[20];
entier util; // taille utile, comme toujours
```

effet en mémoire de la définition du tableau :



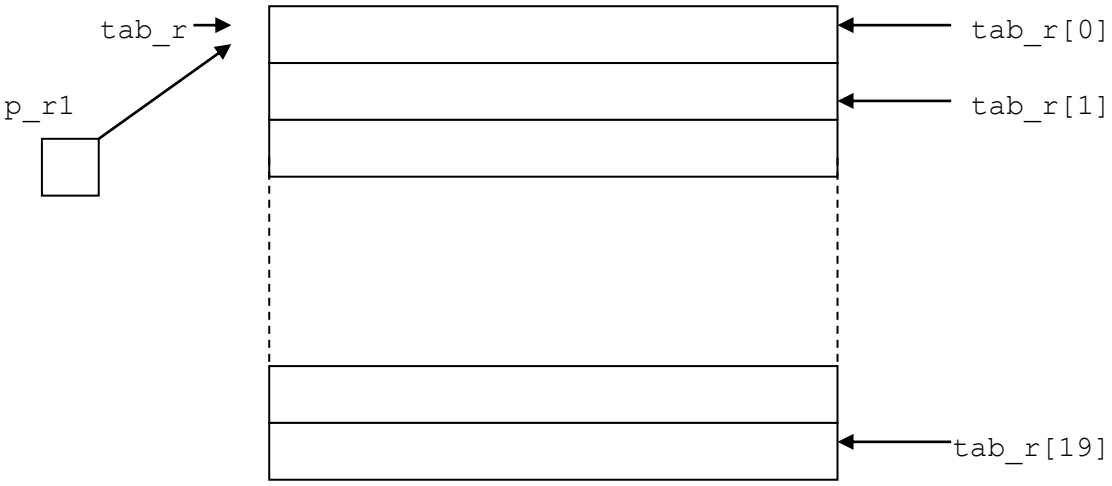
Cette zone où sont stockées les 20 réels est de la forme suivante : (en utilisant un schéma qui est un hybride entre le schéma habituel pour représenter les tableaux et le schéma utilisé pour représenter les zones de mémoire) :



Continuons le programme :

```
reel p_r1;

p_r1 ← tab_r;
```

on affecte au pointeur le tableau : c'est une adresse (l'adresse de base du tableau) qui est recopiée dans le pointeur, ce que l'on schématise de la manière suivante :



le pointeur `p_r1` et le tableau `tab_r` jouent exactement le même rôle, et on peut appliquer indifféremment les notations `*` et `[]` indifféremment à `p_r1` et `tab_r` !

notations équivalentes :

`*p_r1`, `*tab_r`, `p_r1[0]` et `tab_r[0]` sont 4 notations qui permettent d'accéder à la valeur d'indice 0 (située au début de la zone mémoire).

Pour accéder à l'élément d'indice 2, on a encore une fois 4 notations équivalentes :

```
*(p_r1+2)
*(tab_r+2)
p_r1[2]
tab_r[2]
```

En général, pour accéder à la valeur (ou élément) d'indice `i`, `i` étant un indice valide, on peut utiliser l'une des 4 notations :

```
*(p_r1+i)
*(tab_r+i)
p_r1[i]
tab_r[i]
```

En résumé : lorsque l'on manipule une zone mémoire allouée statiquement (par une définition de tableau) ou dynamiquement (par l'intermédiaire de la commande `réservation`), on peut utiliser les notations `*` ou `[]`, qui sont équivalentes.

Retour sur la libération de mémoire

Même si les tableaux et les pointeurs permettent de manipuler des zones de mémoire, celles-ci ne sont pas créées de la même manière. La mémoire allouée pour les tableaux est donnée par le compilateur, sans utiliser l'instruction `reservation` : c'est **une allocation statique**.

La mémoire allouée pour les pointeurs est **allouée dynamiquement** par l'intermédiaire de l'instruction `reservation`.

On n'utilise pas l'instruction `liberer` pour les allocations statiques (tableaux), mais pour les allocations dynamiques (pointeurs).

tableaux à plusieurs dimensions et pointeurs

L'analogie entre tableaux et pointeurs permet même de remplacer indifféremment ces termes pour traiter les tableaux à plusieurs dimensions.

Nous avons évoqué, lors du cours relatif aux tableaux, les tableaux à *N* dimensions, en prenant comme exemple *N*=2, ce qui permet de généraliser aux cas des valeurs supérieures de *N* sans problème.

Un tableau à 2 dimensions est en réalité un tableau de tableaux. On peut donc affirmer qu'il s'agit également d'un :

- Pointeur de tableaux
- Tableau de pointeurs
- Pointeur de pointeurs

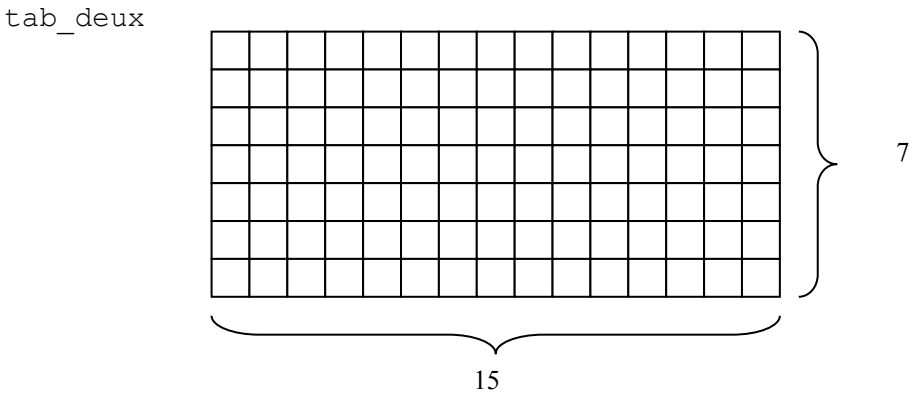
La dénomination "pointeur de tableaux" est en pratique très peu utilisée, mais les autres le sont beaucoup plus.

Représentation avec des adresses (schéma associé aux pointeurs).

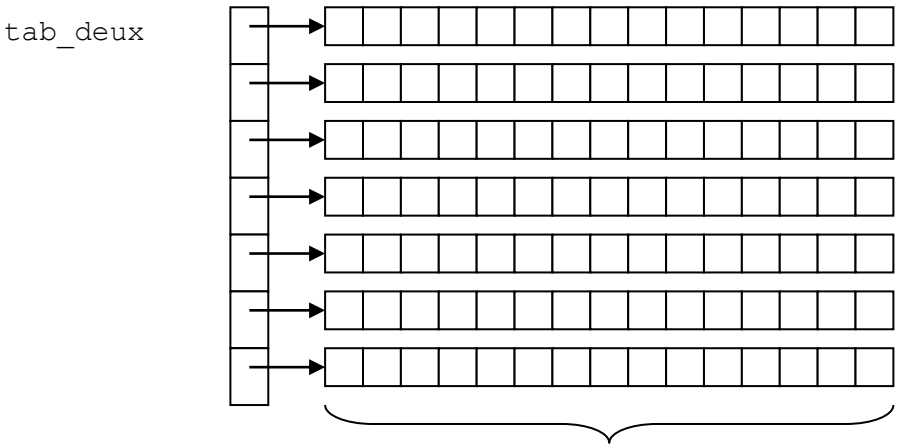
Soit `tab_deux` un tableau à 2 dimensions, contenant, par exemple, des entiers (en fait, le raisonnement qui va suivre est strictement indépendant du type des valeurs contenues dans le tableau).

```
entier tab_deux[7][15]; // exemple de définition
```

Nous avons pris l'habitude de le représenter de la manière suivante, un tableau à 7 lignes et 15 colonnes :

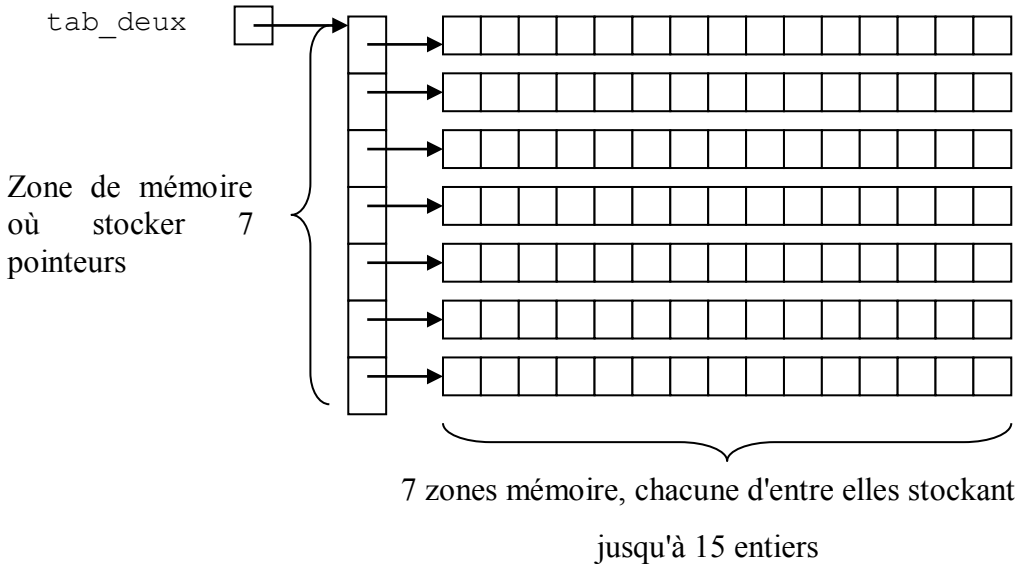


Chaque ligne étant un tableau, il s'agit en fait d'une adresse, donc d'un pointeur. La représentation pour un tableau dont chaque élément est un pointeur est la suivante :



`tab_deux` est également un tableau à 7 éléments, et chacun de ces éléments est un pointeur vers une zone pouvant stocker 15 valeurs entières.

Enfin, puisque `tab_deux` est un tableau, `tab_deux` est un pointeur (seule contrainte : il ne peut pas être modifié) désignant une zone où stocker 7 pointeurs.



Toutes ces représentations sont équivalentes, et correspondent à la définition du tableau à deux dimensions. Cependant, les dénominations "tableaux de pointeurs" et "pointeur de pointeur" peuvent également amener d'autres définitions de variables…

allocation à plusieurs dimensions

Il est possible de définir et de créer de manière totalement dynamique un tableau à 2 dimensions, de manière à n'utiliser en mémoire que la place strictement nécessaire au stockage de l'information utile.

Pour ce faire, considérons le tableau comme un pointeur de pointeur.

Qu'est-ce qu'un pointeur de pointeur ?

Un pointeur de pointeur n'est jamais qu'une variable stockant une adresse, et le contenu de cette adresse est de nouveau une adresse.

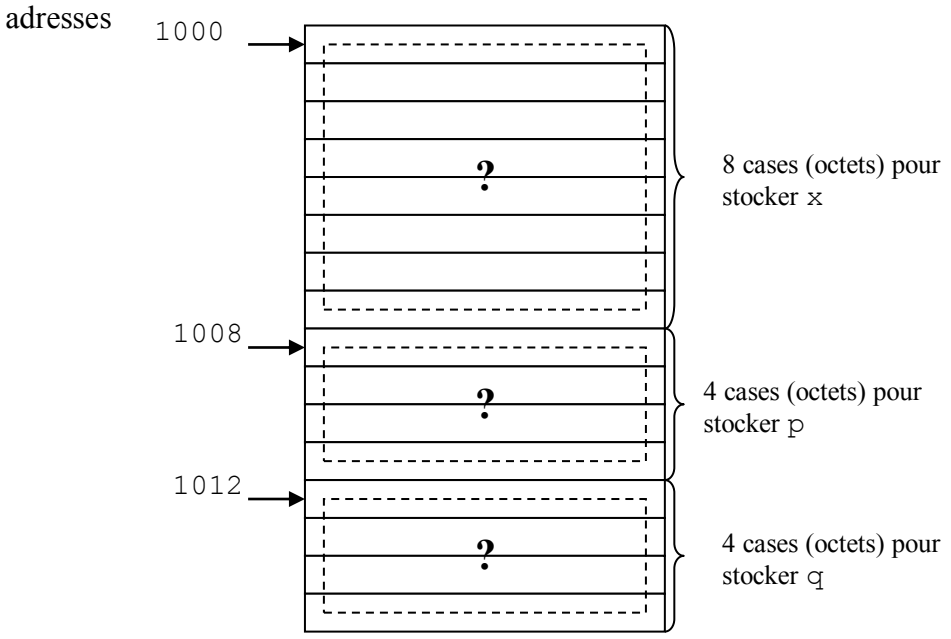
Soit `p` un pointeur, vers un `reel` par exemple. P est également une variable, et comme toutes les variables, lors de sa définition, on lui attribue un nom et un type (éventuellement une valeur). Comme pour toutes les variables, l'ordinateur doit stocker sa valeur en mémoire à une certaine adresse. Un pointeur a donc également une adresse.

Illustration :

```
programme point_point

reel x;
reel *p;
reel **q;
```

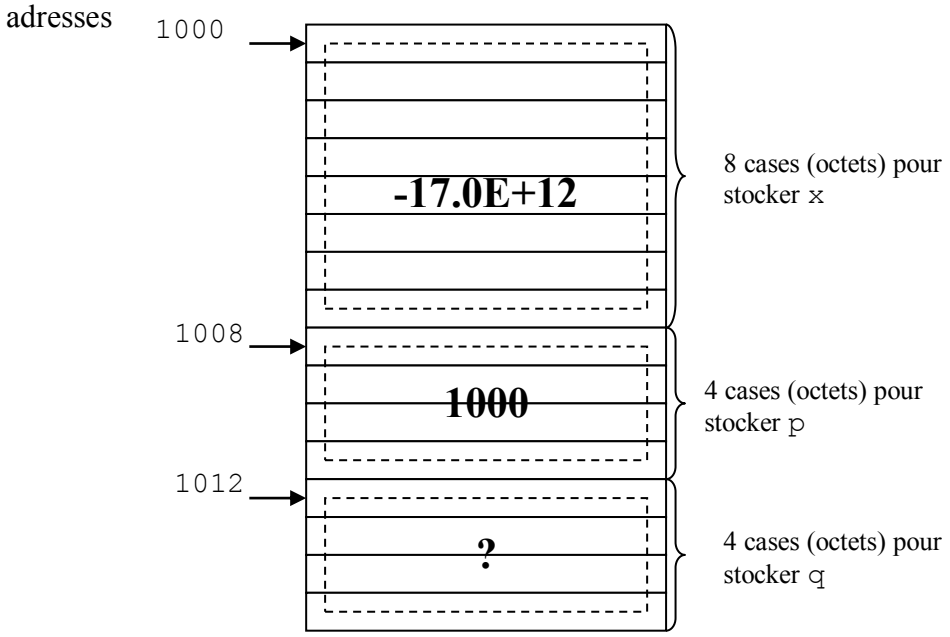
Supposons que le programme stocke ses variables à partir de l'adresse 1000 :

`x` sera stockée sur 8 octets (c'est la taille correspondant au type `double` du C avec lequel est traduit le type `reel` du langage algorithmique), `p` sera stocké sur 4 octets (car un pointeur est codé sur 4 octets), ainsi que `q`.

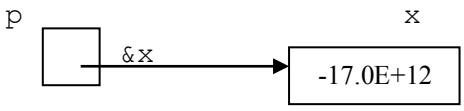


continuons le programme et regardons les effets des affectations suivantes :

```
x ← -17.0E+12;
p ← &x; // d'après les hypothèses, &x vaut 1000, donc p reçoit 1000
```



ce qui correspond également à :

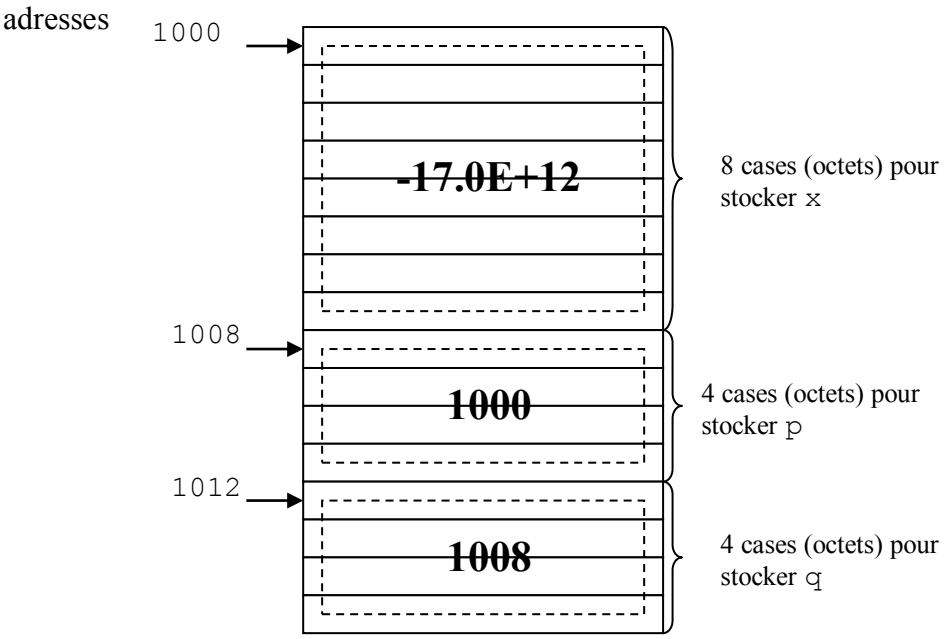


quel est l'effet de l'instruction suivante ?

```
q ← &p; // cette instruction est-elle valable ?
```

cette instruction est valable, car `&p` est l'adresse d'une variable qui stocke une adresse, et `q` stocke l'adresse d'une adresse.

`&p` vaut 1008 (d'après le schéma) et donc c'est cette valeur 1008 qui est affectée à `q`.



pointeurs et tableaux de caractères

Un tableau de caractères est donc analogue à un pointeur vers caractère, comme nous venons de le voir. Ainsi, un texte peut être stockée dans une zone de mémoire dont l'adresse peut être stockée dans un pointeur, si l'on veut effectuer des allocations dynamiques. Or cette situation se reproduit très souvent dans les programmes, tout simplement parce qu'une information que l'on garde sous la forme d'un texte présente la propriété de pouvoir être de longueur variable.

Prenons l'exemple d'un programme qui a pour objet de gérer une base de données contenant des informations sur des personnes : dans un cas simple, on voudra stocker son age et son nom. Pour l'age, une variable de type `entier` est largement suffisante (on peut même utiliser le type `caractere`). Par contre, pour le nom, on doit utiliser du texte, donc un tableau de `caractere`, dont la longueur peut être variable. Même si la majorité des noms sont relativement courts (moins de 20 caractères), si l'on utilise un tableau, il est nécessaire de prévoir les cas au pire, c'est à dire 50 ou 60 caractères. Cela peut générer une mauvaise utilisation de la mémoire : pour une base de données de 10 000 personnes, on devra donc utiliser 500 000 caractères, alors que l'on n'en utilisera réellement que 30 à 40 %. Il est donc naturel d'utiliser les pointeurs et l'allocation dynamique pour la gestion de texte.

Les commandes de gestion du texte que nous avons rencontrées dans le cours concernant les tableaux fonctionnent également avec les pointeurs vers caractère, puisque, à peu de choses près, ces types sont équivalents.

Déterminer le nombre de caractères nécessaires au stockage du texte : ceci est possible grâce à l'utilisation de la commande `longueur_texte`. Mais cette commande ne s'applique qu'à un texte existant, donc déjà saisi…Ce qui signifie que le tableau de `caractere` nécessaire au stockage du texte doit être défini avant la saisie !

Lorsque l'on veut créer un texte avec la longueur adéquate, il suffit en fait de disposer de deux zone de mémoire : une qui sera assez grande pour accueillir tous les textes possibles (disons, 1000 caractère, ce qui pour une saisie au clavier est largement suffisant). Nommons cette zone `txt_tempo`. La saisie s'effectuera dans cette zone.

Comment saisir ou afficher du texte avec un pointeur vers des caractères ?

Les tableaux et les pointeurs étant analogues, les commandes `lire` et `ecrire` sont incapables (et heureusement d'ailleurs) de distinguer du texte stocké dans un tableau ou dans une zone de mémoire allouée dynamiquement !

Une fois la saisie dans ce tableau faite, on peut récupérer la longueur utile du texte et créer une seconde zone, nommée `txt_final`, qui aura la taille adéquate pour stocker uniquement les caractères valides. Le texte contenu dans `txt_tempo` est alors recopié dans la zone `txt_final`.

Et la zone `txt_tempo` ? il suffit maintenant de la libérer avec la commande…`liberer`, et le tour est joué !

Illustration :

```
programme saisie_utile

caractere *txt_tempo;
caractere *txt_final;
entier utile;

txt_tempo ← reservation(1000 caractere); // comme si on avait un
                                         // tableau de 1000 caracteres
afficher("saisissez votre texte :");
lire(txt_tempo); // aucun problème, cela fonctionne
```

```
utile ← longueur_texte(txt_tempo); // nombre de caractères utiles
utile ← utile+1; // car longueur_texte ne compte pas le caractere
                 // EOT qui doit être recopié

txt_final ← reservation(utile caractere); // et voilà la zone a la
                                          // bonne taille
copier(txt_final, txt_tempo); // recopie des caractères

liberer(txt_tempo); // et voilà, les 1000 caracteres sont restituées

afficher("vous avez saisi : ");
ecrire(txt_final);
```