*École de technologie supérieure*

# THÈME 9
# La magie des pointeurs et les petits biscuits [ ]

INF125 Introduction à la programmation
Sylvie Ratté et Hugues Saulnier

``Pointers are like jumps, leading wildly from one part of the data structure to another. Their introduction into high-level languages has been a step backwards from which we may never recover'' C.A.R.Hoare ``Hints on Programming Language Design'' 1973

## 1. Pourquoi le & lorsqu'on utilise "scanf"

### A.1. Le type "adresse de" ou "Variation sur le thème des tableaux passés en paramètres"

Considérez le petit programme suivant:

```
void exemple1(double P[]);
int main(void){
  double T[15];
  exemple1(T);
  printf("et nous avons %f\n", T[0]);

  return EXIT_SUCCESS;
}

void exemple1(double P[]){
   P[0] = 942;
}
```

Nous savons tous que le tableau T sera modifiée suite à l'appel de la fonction. Autrement dit, le contenu du tableau T (plus particulièrement, celui de l'emplacement 0) n'est pas le même avant et après l'appel de la fonction.

Ce comportement est-il curieux? Que se passe-t-il lorsque nous utilisons une variable ordinaire.

```
void exemple2(double P);
int main(void){
  double X;
  exemple2(X);
  printf("et nous avons %f\n", X);

  return EXIT_SUCCESS;
}

void exemple2(double P){
   P = 942;
}
```

Pourquoi dans ce cas, la variable "X" n'est pas modifiée comme le serait le tableau T examiné précédemment? Autrement dit, quelle différence y a-t-il entre les deux fonctions "exemple":

| | |
|---|---|
| `void exemple1(double P[]){`<br>`  P[0] = 942;`<br>`}` | `void exemple2(double P){`<br>`  P = 942;`<br>`}` |
| appel:<br>`  exemple1(T);`<br>où T est déclarée: `double T[15];` | appel:<br>`  exemple2(X);`<br>où X est déclarée: `double X;` |

La raison est simple, les crochets du tableau [ ] sont en fait un artifice qui cache une réalité des plus crue. Lorsque nous passons un tableau en paramètre, nous passons en fait L'ADRESSE DU PREMIER EMPLACEMENT (l'emplacement zéro) du tableau. Comment pourrait-on imiter ce comportement avec une variable ordinaire? Il suffit de passer son adresse et nous savons, parce que nous utilisons "scanf" depuis le début, que cette adresse est accessible grâce à l'opérateur &. Il suffirait donc de faire ce qui suit dans notre:

```
exemple2(&X);
```

Mais quel serait le type du paramètre de la fonction? La valeur que nous allons recevoir se décrit comme suit: il s'agit de l'ADRESSE D'UN EMPLACEMENT QUI PEUT CONTENIR UN DOUBLE. On note cela comme suit: double * tout simplement. Le type du paramètre est donc "double *" et non pas "double". Le prototype de la fonction "exemple" devient:

```
void exemple2(double * P);
```

Les deux types sont bien distincts. Pour le vérifier, écrivez un petit programme qui permet d'afficher la taille des deux types avec "sizeof". sizeof(double) retourne 8 tandis que sizeof(double *) retourne 2 ou 4 (selon le système utilisé).

## A.2. Accès à l'emplacement d'origine

Considérez une fois de plus le premier programme qui manipule un tableau. Est-ce que la fonction "exemple1" effectue quelque chose de particulier sur son paramètre. À première vue non. Cependant, sachez que les crochets de tableau camouflent le fait que T est en fait de type "double *". Comment cela se peut-il? Rappelez-vous que T contient en fait:L'ADRESSE DU PREMIER EMPLACEMENT. Quel est le type de cet emplacement? Réponse: "double". DONC, T est bien de type "double*". Lorsque nous passons notre tableau en paramètre, nous ne passons pas une copie des 100 emplacements auxquels la variable réfère mais bien une copie de l'adresse du premier emplacement. Ainsi, lorsque le compilateur rencontre M[i], il effectue simplement un décalage à partir de l'adresse de départ contenue dans M. On a ainsi accès, par un simple artifice, à un emplacement qui n'appartient pas à la fonction mais bien à un autre bloc (le bloc qui a appelé la fonction, ici il s'agit du "main").

Ainsi, nous pourrions réécrire notre fonction "exemple1" de la manière suivante:

```
void exemple1(double * P){
   P[0] = 942;
}
```

Mais comment aurons-nous accès à la variable "x" du "main" au travers son adresse. On pourrait, même si conceptuellement c'est un peu curieux, utiliser le même "truc" et ajouter des [ ] à notre paramètre P. On obtiendrait ici le comportement désiré puisque lorsque le compilateur rencontre P[0] il effectuerait simplement un décalage à partir de l'adresse de départ contenue dans P. Puisque le décalage est zéro, on a ainsi accès à la variable "x"! Évidemment, nous ne pourrions pas utiliser P[3] ou p[100] ou même P[1] car dans ce cas, on déborderait immédiatement...Autrement dit,

```
void exemple2(double * P);
int main(void){
  double X;
  exemple2(&X);
  printf("et nous avons %f\n", X);

  return EXIT_SUCCESS;
}

void exemple2(double * P){
   P[0] = 942;
}
```

Cependant cette manière de procéder n'est pas très orthodoxe. En général, on la réserve à l'utilisation des "vrais" tableaux. On utilise l'opérateur * devant une variable contenant une adresse pour indiquer que nous désirons avoir accès à l'emplacement désigné par l'adresse. Notre fonction devient:

```
void exemple2(double * P);
int main(void){
  double X;
  exemple2(&X);
  printf("et nous avons %f\n", X);

  return EXIT_SUCCESS;
}

void exemple2(double * P){
   *P = 942;
}
```

On nomme POINTEURS les variables qui contiennent des ADRESSES bien typées. Ainsi, toutes les variables suivantes sont des pointeurs:

```
double * P;
int * Q;
unsigned long * R;
char * S;
```

et le tableau suivant contient 50 pointeurs puisque chacun de ces emplacements est de type "adresse de".

```
char * T[50];
```

Où sont les 50 emplacements "char" ainsi pointés? Bonne question. Pour l'instant, ils ne sont pas créés! Nous verrons comment effectuer cette tâche simple à la section 3.

## 2. Pointeurs, tableaux

### A. Arithmétiques de pointeurs

Puisque notre variable T, notre tableau précédent, contient en fait une adresse. Que se passe-t-il si j'effectue:

```
*T = 567.5;
```

Cette assignation est tout à fait légale puisque T est une adresse. Où sera placée la valeur 567.5? Dans le premier emplacement du tableau bien sûr! Ainsi donc, T[0] contiendra 567.5.

Que se passe-t-il si j'effectue:

```
*(T+1) = 888.7;
```

C'est ici que ça devient intéressant. T contient une adresse, donc un chiffre. On additionne 1 à ce chiffre et on traite le résultat comme une nouvelle adresse. Mais de quel emplacement s'agit-il? Se retrouve-t-on 1 bit plus loin, 1 octet plus loin, ou quoi encore?

En fait, le système C est assez intelligent pour évaluer ce 1 selon le contexte de son utilisation. Puisque T est de type "double *", il réfère donc à un emplacement de type "double". Le +1 aura l'effet d'additionner à T la valeur de sizeof(double). Donc T+1 nous amène 8 octets plus loin. DIRECTEMENT SUR L'EMPLACEMENT [1] du tableau!

En conséquence: *(T+5) est l'équivalent de T[5] et *(T+2) est l'équivalent de T[2], et ainsi de suite.

Voici deux versions d'un bloc de code effectuant l'affichage du contenu d'un tableau. Le premier utilise les crochets et le second utilise l'arithmétique de pointeurs.

CODE avec petits biscuits [ ]

```
void affiche1(long tab[]){
  int i;
  for (i = 0; i < TAILLE; i++)
    printf("%li ", tab[i]);
}
```

CODE avec gymnastique arithmétique de pointeurs

```
void affiche2(long * tab){
  long * ptr;
  for (ptr = tab; ptr < tab+TAILLE; ptr++)
    printf("%li ", *ptr);
}
```

Voici maintenant deux versions d'un bloc de code effectuant l'affichage inverse du contenu d'un tableau.

CODE avec petits biscuits [ ]

```
void afficheInv1(long tab[]){
  int i;
  for (i = TAILLE-1; i >= 0; i--)
    printf("%li ", tab[i]);
}
```

CODE avec gymnastique arithmétique de pointeurs

```
void afficheInv2(long * tab){
  long * ptr;
  for (ptr = tab+TAILLE-1; ptr >= tab; ptr--)
    printf("%li ", *ptr);
}
```

NOTE: Seule les additions et les soustractions sont permises sur les pointeurs.

## B. Retour sur les fonctions

Considérez la fonction "permute" que nous avons utilisé pour illustrer quelques tris au thème 7.

CODE de permute

```
void permute(double * A, double * B){
  double tampon;
  tampon = *A;
  *A = *B;
  *B = tampon;
}
```

CODE du tri par insertion (NOTEZ BIEN L'UTILISATION DE LA FONCTION "permute")

```
void triInsertion(double T[],int a, int b)
{  int i;
   int k;
   for(i = a ; i < b; i++)
   {  for (k = i; k > a; k--){
        if (T[k] < T[k-1]) permute(&T[k],&T[k-1]);
      }
   }
}
```

Ce code pourrait être réécrit comme suit:

CODE du tri par insertion (NOTEZ BIEN L'UTILISATION DE LA FONCTION "permute")

```
void triInsertion(double T[],int a, int b)
{  int i;
   int k;
   for(i = a ; i < b; i++)
   {  for (k = i; k > a; k--){
        if (T[k] < T[k-1]) permute(T+k,T+k-1);
      }
   }
}
```

## C. Petit jeu de parasitage de pointeurs

Considérez le petit programme suivant dans lequel les variables P, Q, R et PT parasitent des variables simples.

```
int main(){
  long X; long Y; long W; long T;
  long * P; long * Q; long * R; long * PT;
  X = 350; Y = 25; W = 5;
  P = &X; Q = &Y; R = &W;
  /* affichage des valeurs */
  printf("X = %4li Y = %4li W = %4li *P = %4li *Q = %4li *R = %4li\n", X, Y, W, *P, *Q, *R);
  /* échange de valeurs */
  T = W; W = Y; Y = X; X = T;
  /* affichage des valeurs */
  printf("X = %4li Y = %4li W = %4li *P = %4li *Q = %4li *R = %4li\n", X, Y, W, *P, *Q, *R);
  /* échange des pointeurs (adresses) */
  PT = R; R = Q; Q = P; P = PT;
  /* affichage des valeurs */
  printf("X = %4li Y = %4li W = %4li *P = %4li *Q = %4li *R = %4li\n", X, Y, W, *P, *Q, *R);
  return 0;
}
```

Quel est le résultat de l'exécution de ce petit programme?

```
X =  350 Y =   25 W =    5 *P =  350 *Q =   25 *R =    5
X =    5 Y =  350 W =   25 *P =    5 *Q =  350 *R =   25
X =    5 Y =  350 W =   25 *P =   25 *Q =    5 *R =  350
```

# 3. Allocation dynamique et copie de blocs de mémoire

## A. Allocation dynamique de la mémoire

### A.1. INTRODUCTION AU CONCEPT

Les tableaux que nous avons vus jusqu'à présent sont relativement limités. Le problème vient du fait que leur taille doit être connue au moment de la compilation. Si l'espace est trop grand, il en résulte une perte d'espace mémoire lors de l'exécution et si l'espace est trop petit, le programme devient inutilement limité ou ne fonctionnera pas (si la validation des indices du tableau n'est pas effectuée correctement). Il serait intéressant, pour contrecarrer cette limite, de pouvoir utiliser une variable contenant la taille éventuelle du tableau. On pourrait ainsi déterminer la taille au moment de l'exécution et non au moment de la compilation. Imaginons un instant ce que l'on pourrait faire si le C nous permettait la chose. Le problème pourrait consister, par exemple, à faire certains traitements sur un ensemble de notes d'étudiants alors que le nombre n'est pas connu à l'avance.

```
int main(){
  int taille;
  printf("Combien de notes devez-vous saisir? ");
  scanf("%i", &taille);
  double notes[taille];
  ...
}
```

Malheureusement cette méthode ne fonctionne pas en C. D'abord, la variable tableau "notes" se trouve déclarée en plein milieu d'un bloc et non au début. Ensuite, le compilateur doit absolument connaître le nombre d'octets à réserver au moment de la compilation.

Cependant, il existe une façon de contourner le problème: l'allocation dynamique de la mémoire. On dit "dynamique" par opposition à "statique" parce que ce type d'allocation sera effectuée lors de l'exécution du programme et non lors de sa compilation. Pour effectuer une allocation dynamique, on utilise les fonctions "malloc" ou "calloc".

### A.2. MALLOC ET LA VIANDE CRUE

La fonction "malloc" appartient à la librairie "stdlib". Sa syntaxe d'utilisation est la suivante:

```
malloc(taille en octets du bloc mémoire désiré)
```

La fonction retourne un pointeur sur le 1er emplacement alloué. Si elle ne réussit pas à effectuer son travail (mémoire non disponible!), elle retourne la constante NULL (c'est-à-dire zéro). L'avantage ici c'est que l'argument n'a pas à être une constante.

Mais quel type de pointeur retourne-t-elle? Un pointeur sur un int? un double? un char?

En fait, comme la fonction ne sait pas comment nous allons utiliser la série d'octets qu'elle aura réservée, elle retourne ce que nous appelons "un pointeur générique". Il s'agit d'un pointeur de type (void *). Ce pointeur (void *) doit être interprété comme "un pointeur sur n'importe quoi" (!) et non comme "un pointeur sur rien"! On devrait toujours convertir explicitement le résultat retourné par "malloc" avant de l'utiliser.

EXEMPLE

Supposons que nous désirions avoir accès à une série de 1500 emplacements "double", on fera:

```
double * T;
T = (double *) malloc(1500 * sizeof(double));
```

Comment pourrons-nous utiliser ces emplacements?

Nous pourrions utiliser des décalages (arithmétique de pointeurs) à partir de l'adresse de départ:

- `*T` nous donne accès au 1er emplacement
- `*(T+1)` nous donne accès au 1er emplacement
- `*(T+2)` nous donne accès au 2e emplacement

et ainsi de suite jusqu'à

- `*(T+1499)` qui nous donne accès au 1500e emplacement

Nous pourrions également utiliser les agréables crochets de tableaux:

- `T[0]` nous donne accès au 1er emplacement

`T[1]` nous donne accès au 1er emplacement
`T[2]` nous donne accès au 2e emplacement
et ainsi de suite jusqu'à
`T[1499]` qui nous donne accès au 1500e emplacement

Contrairement aux tableaux statiques qui se retrouvent dans la pile ou dans le segment des données (si le mot "static" précède leur déclaration), la mémoire allouée à un tableau dynamique se retrouve ailleurs dans le RAM. On n'est donc pas limité par l'espace utilisé par le processus. On est plutôt limité par la quantité de mémoire disponible (et libre) sur notre machine. C'est en fait le système d'exploitation (SE) qui veillera à répondre aux demandes de "malloc".

Évidemment, il y a un certain coût associé à l'utilisation de "malloc". Comme le SE doit garder la trace de toute allocation mémoire, il utilise un bout de la mémoire pour se rappeler des données essentielles (longueur du bloc alloué, adresse de départ, etc.). C'est pourquoi, on utilise les allocations dynamiques uniquement lorsque nos blocs sont assez gros ou lorsque le traitement le justifie (nous en reparlerons).

### A.3 LIBÉREZ LA MÉMOIRE

Lorsque vous avez terminé d'utiliser la mémoire, n'oubliez jamais de libérer celle-ci! On effectue cette tâche grâce à la fonction "free". Elle s'utilise comme suit:

free(*pointeur sur le 1er emplacement*);

La libération de la mémoire est ESSENTIELLE sans quoi la mémoire devient de plus en plus occupée par des blocs inutiles. En général, on fera la désallocation (le free) dans le même bloc de code où se trouve l'allocation (malloc ou calloc). Dans toutes les circonstances, il faut simplement s'assurer que le pointeur sur le 1er emplacement sera encore disponible lorsque s'effectuera le "free".

Voici, en guise d'illustration une fonction qui a besoin d'un tableau dynamique pour effectuer une certaine tâche. Si le "free" n'est pas effectué, la variable P disparaît à la fin de l'exécution de la fonction mais les 1500 doubles parasitent la mémoire. Le pointeur sur le premier emplacement étant perdu, ce gros bloc de mémoire est désormais inaccessible pendant tout le reste de l'exécution de notre programme.

```
int main(void){
   exemple1();
   ... les 1500 emplacements sont désormais détruits ...
}
void exemple1(void){
   double * T;
   T = (double *) malloc(1500 * sizeof(double));
   ... traitement sur les 1500 emplacements ...
   free(T);    /* ABSOLUMENT ESSENTIEL DANS CE CONTEXTE */
}
```

Par contre, si la fonction retourne le pointeur au bloc de code qui l'a appelé, le pointeur reste connu et l'on pourra effectuer le "free" ailleurs dans le programme.

```
int main(void){
   double * P;
   P = exemple2(); /* on récupère le pointeur */
   ... suite du traitement ...
   free(P);
   ... les 1500 emplacements sont désormais détruits ...
}
double * exemple2(void){
   double * T;
   T = (double *) malloc(1500 * sizeof(double));
   ... traitement sur les 1500 emplacements ...
   return T;
}
```

On pourrait donc tester facilement l'espace mémoire disponible sur votre ordinateur en utilisant une petite boucle (il existe une autre manière plus commode de récupérer cette information):

```
int NbBlocs = 0;
void * R;
NbBlocs = 0;
do{
  if ((R = malloc(1024)) != NULL) NbBlocs++;
}while (R != NULL);
printf("%i Ko alloués!\n", NbBlocs);
```

### A.4. CALLOC ET REALLOC

Tout comme "malloc", la fonction "calloc" permet d'allouer un bloc mémoire comme suit:

```
calloc(nb-éléments, taille-de-chaque-élément);
```

ce qui est l'équivalent de:

```
malloc(nb-éléments * taille-de-chaque-élément);
```

La seule différence est que la fonction "calloc" initialise à ZÉRO les emplacements alloués, ce que n'effectue jamais la fonction "malloc".

Quant à la fonction "realloc", elle permet de réallouer de la mémoire en conservant les valeurs qui se trouvaient originalement dans le bloc mémoire. Le nouveau bloc mémoire peut être plus grand ou plus petit que l'original. Cette fonction permet ainsi "d'étirer" ou "de raccourcir" un bloc mémoire selon nos besoins. On l'utilise comme suit:

```
realloc(pointeur-sur-le-bloc-à-modifier, taille-en-octets-désirée);
```

## B. Copie dynamique de blocs de mémoire

### B.1. COPIE ET COMPARAISON DE BLOCS DE MÉMOIRE

Prenez deux tableaux statiques T et P déclarés comme suit:

```
double T[1500];
double P[1500];
```

Sur ce type de tableau, l'assignation est illégale (cet aspect est examiné à la section C plus bas), de sorte que si l'on désire copier le contenu de P dans T, ON NE PEUT ABSOLUMENT PAS utiliser:

```
T = P;
```

Dans le même ordre d'idée, si l'on désire comparer le contenu de T avec celui de P, le test suivant ne fera pas l'affaire puisque dans ce cas, on testerait l'égalité des deux adresses et non l'égalité des deux contenus (1500 emplacements comparés un à un).

```
if (T == P) ...
```

La fonction "memcpy" permet justement de copier des blocs mémoire de manière très rapide tandis que la fonction "memcmp" permet de comparer le contenu de deux blocs mémoire.

Ces deux fonctions s'utilisent comme suit:

```
memcpy(pointeur-sur-bloc-1, pointeur-sur-bloc-2, nb-d'octets-à-copier);

memcmp(pointeur-sur-bloc-1, pointeur-sur-bloc-2, nb-d'octets-à-comparer);
```

La fonction "memcpy" copie les "nb" premiers octets de "bloc-2" sur les "nb" premiers octets de "bloc-1". On peut utiliser cette fonction uniquement si les deux blocs ainsi visés ne se recoupent pas.

La fonction "memcmp" retourne un entier négatif si le contenu du "bloc-1" est plus petit que le contenu du "bloc-2" , un entier positif si le contenu du "bloc-1" est plus grand que le contenu du "bloc-2" et zéro, si le contenu du "bloc-1" est égal au contenu du "bloc-2".

Ainsi, pour copier le contenu de P dans T, on fera:

```
memcpy(T, P, 1500 * sizeof(double));
```
ou (puisque P est un tableau statique)
```
memcpy(T, P, sizeof(P));
```

On pourrait évidemment copier un bout de tableau dans un autre, sans itération:

```
memcpy(T, P, 5*sizeof(double));
memcpy(T+10, P+5, 40*sizeof(double));
```

Comme d'habitude, il faut évidemment être certain de ne pas déborder de l'espace originalement alloué!

Pour comparer les deux tableaux, on devra donc tester le résultat de "memcmp":

```
r = memcmp(T, P, 1500 * sizeof(double));
if (r < 0)
  ...contenu de T < que contenu de P
else if (r > 0)
  ...contenu de T > que contenu de P
else
  ...contenu de T == contenu de P
```

### B.2. DÉPLACEMENT D'UN BLOC DE MÉMOIRE

La fonction "memmove" copie les "nb" premiers octets du "bloc-2" sur les "nb" p

```
memmove(pointeur-sur-bloc-1, pointeur-sur-bloc-2, nb-d'octets-à-déplacer);
```

### B.3. INITIALISATION D'UN BLOC DE MÉMOIRE

La fonction "memset"

```
memset(pointeur-sur-bloc, valeur-initiale-sur-un-octet, nb-d'octets-à-initialiser);
```

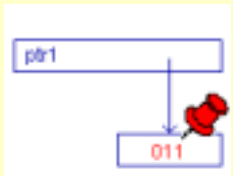
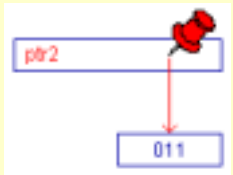
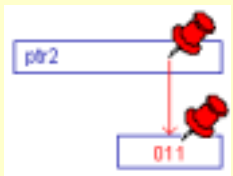
### B.4. RECHERCHE DANS UN BLOC DE MÉMOIRE

La fonction "memchr"

```
memchr(pointeur-sur-bloc, valeur-sur-un-octet-à-chercher, nb-d'octets-à-rechercher);
```

## C. Tableaux, pointeurs et const

Puisque nous pouvons déclarer des variables entières ou réelles comme des constantes, nous pouvons également avoir des pointeurs constants (inamovibles). Mais comme nous avons affaire à deux emplacements (l'emplacement contenant l'adresse et l'emplacement pointé), nous pouvons mettre le "const" à deux niveaux. Le tableau suivant résume les possibilités.

| | | |
|---|---|---|
| `const int * ptr1 = &a;` | L'entier pointé par "ptr1" est "gelé". L'instruction suivante devient illégale:<br>**`*ptr1 = 3;`**<br>Par contre, le pointeur lui-même n'est pas "gelé", l'instruction suivante est donc légale:<br>`ptr1 = &b;` |  |
| `int * const ptr2 = &a;` | L'entier pointé par "ptr2" n'est pas "gelé". L'instruction suivante devient légale:<br>`*ptr2 = 3;`<br>Par contre, le pointeur lui-même est "gelé", l'instruction suivante est donc illégale:<br>**`ptr2 = &b;`** |  |
| `const int * const ptr2 = &a;` | L'entier pointé par "ptr2" est "gelé". L'instruction suivante devient illégale:<br>**`*ptr2 = 3;`**<br>Le pointeur lui-même est "gelé", l'instruction suivante est donc illégale:<br>**`ptr2 = &b;`** |  |

### Polémique autour des tableaux

Énoncé de la polémique:

**Le nom du tableau, c'est l'adresse du tableau.**

À strictement parler, cette affirmation est FAUSSE. De fait, il faut bien comprendre ce qui se passe lorsqu'on utilise un tableau tel que "tab" déclaré comme suit:

```
int tab[200];
```

Lorsqu'on utilise "tab", il y a une opération cachée qui est effectuée et qui fait en sorte que le nom "tab" est automatiquement TRANSFORMÉ en l'adresse du premier octet du tableau. Mais, encore une fois, le nom "tab" n'est pas l'adresse du tableau. Si "tab" était véritablement et immédiatement un pointeur (une adresse), l'expression "sizeof(tab)" retournerai 2 ou 4 (selon la taille des pointeurs), mais cette expression retourne 200.

La transformation automatique du nom du tableau en l'adresse du premier octet permet évidemment les expressions suivantes:

```
tab[0] ou encore *tab
tab[5] ou encore *(tab + 5)
```

Par contre, le "pointeur" ainsi obtenu automatiquement NE PEUT ÊTRE MODIFIÉ comme pourrait l'être un véritable pointeur tel que tab2:

```
int * tab2 = (int *) malloc(200 * sizeof(int));
```

Ainsi, alors que:

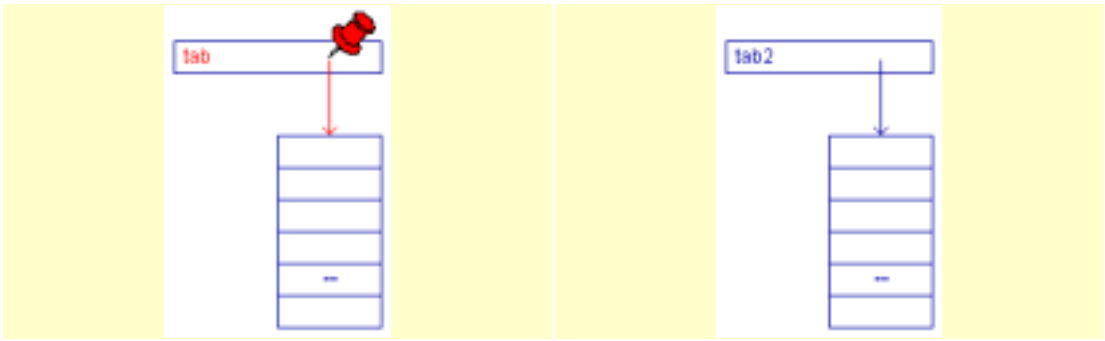
```
tab2 = &a;
```

est tout à fait légale si "a" est de type "int",

```
tab = &a;
```

N'est PAS ADMISSIBLE. En somme, le pointeur obtenu automatiquement et de manière tout à fait transparente, est un pointeur CONSTANT (cf. const). En somme, on pourrait résumer la situation comme suit:

À gauche, le pointeur obtenu automatiquement à partir du nom "tab", à droite, le vrai pointeur "tab2".




Si un utilisateur n'a pas vu la déclaration d'un tableau, la seule façon de savoir auquel des deux il a affaire est d'effectuer un "sizeof".