C

Wieslaw Zielonka
zielonka@irif.fr

Rappel

pointeur == adresse

`&variable` l'adresse d'une variable

`&tab[i]` l'adresse d'un élément de tableau

`int *p_a;` déclaration d'une variable p_a de type pointeur vers int, p_a sert à stocker l'adresse d'une donnée de type int

`int a = 13; int *p_a = &a;` /* mettre dans p_a l'adresse de la variable a */

```
typedef struct{
        double x;
        double y;
}  point;

point p = { .x = 5.5 };

point *p_point = &p;   /* p_point contient l'adresse de la variable p */
```

Rappel

dans une expression à droite de =

*pointeur

donne la valeur de la donnée se trouvent à l'adresse pointeur :

int a = 10;

int *p_a = &a;

```
int b = *p_a * *p_a + *p_a - 1;
```

*p_a == 10   donc   b prend la valeur   10*10+10-1 == 109

Rappel

`*pointeur` à gauche de `=`

`*pointeur = expression;`

```
int a ;

int *p_a = &a;
```

`*p_a = 2 ;` mettre la valeur 2 à l'adresse qui est stockée dans la variable p_a (donc dans a)

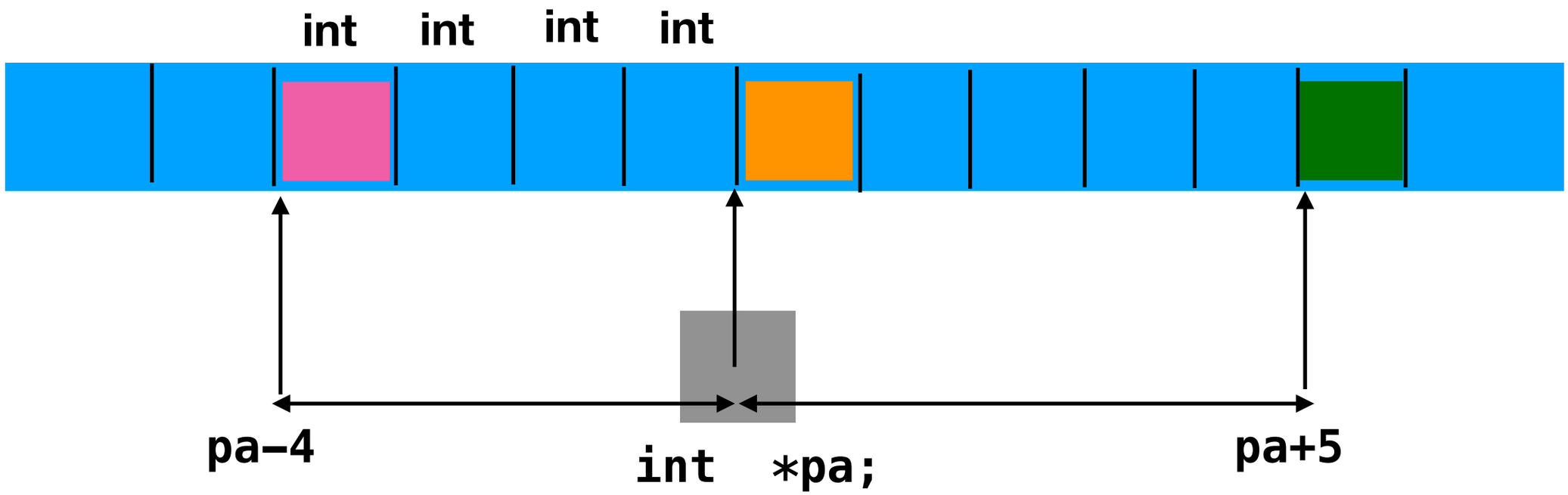
```
*p_a = 2 + 3 * *p_a;
```

prendre un int qui se trouve à l'adresse stockée dans p_a, le multiplier par 3,

ajouter 2 et remettre à l'adresse stockée dans p_a

Rappel

pointeur + entier pointeur - entier **sont aussi des pointeurs**



int

pa contient l'adresse de 

pa-4 l'adresse de 

puisque pa est un pointeur vers int l'adresse pa-4 est calculée en tenant compte de la taille d'un int

Rappel

```
unsigned int tab[]={1,2,3,4,5,6,7,8};

unsigned int *p = &tab[3];

*(p-1) = *(p+1) + *(p+2);
```

prendre un int qui se trouve à l'adresse **p+1** et un int qui se trouve à l'adresse **p+2**, additionner et mettre le résultat à l'adresse **p-1**

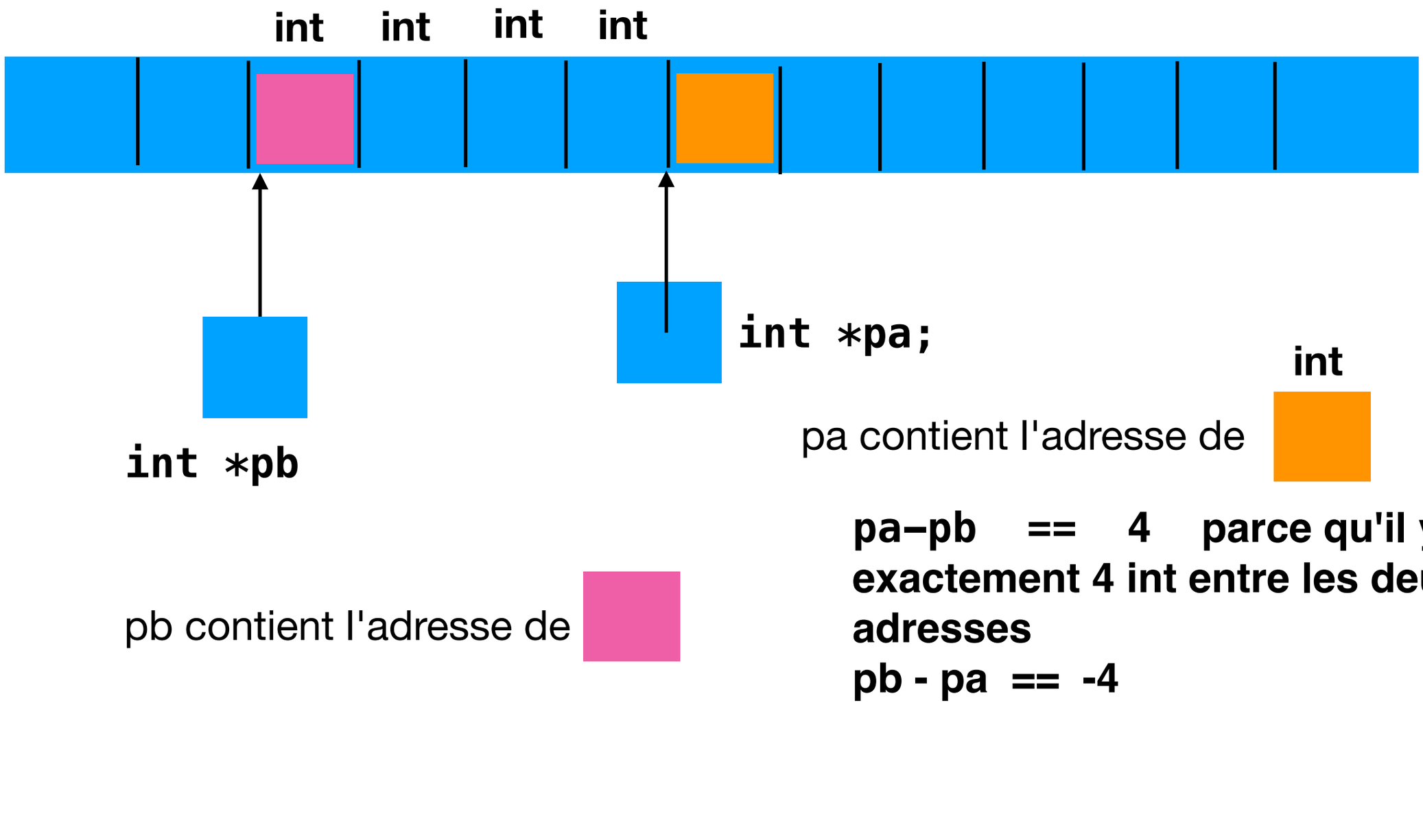
équivalent à : `p[-1] = p[1]+p[2];`

Quel élément du tableau change? Quelle est la nouvelle valeur?

Rappel

pointeur1 - pointeur2

est un entier signé de type `ptrdiff_t`, **les deux pointeurs doivent être de même type**



pa−pb == 4 parce qu'il y a exactement 4 int entre les deux adresses
pb - pa == -4

réduction de tableau vers pointeur lors de l'appel de fonction

```
double somme(int nb_elem, int t[]){  }
```

```
double somme(int nb_elem, int *t){  }
```

équivalent, le compilateur C traduit le paramètre t de la fonction moy() en `int *t`

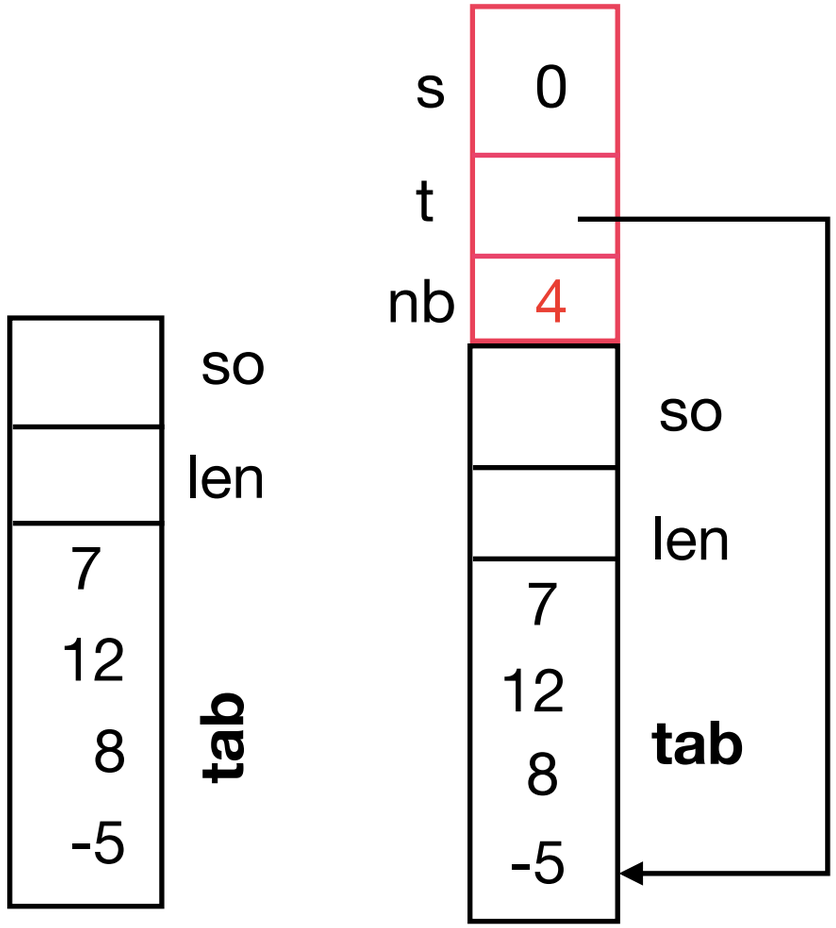
réduction de tableau vers pointeur lors de l'appel de fonction

```
int somme(int nb, int t[]){
  int s = 0;
  for( int i = 0; i < nb; i++){
    s += *t;
    t++;
  }
  return s;
}
```

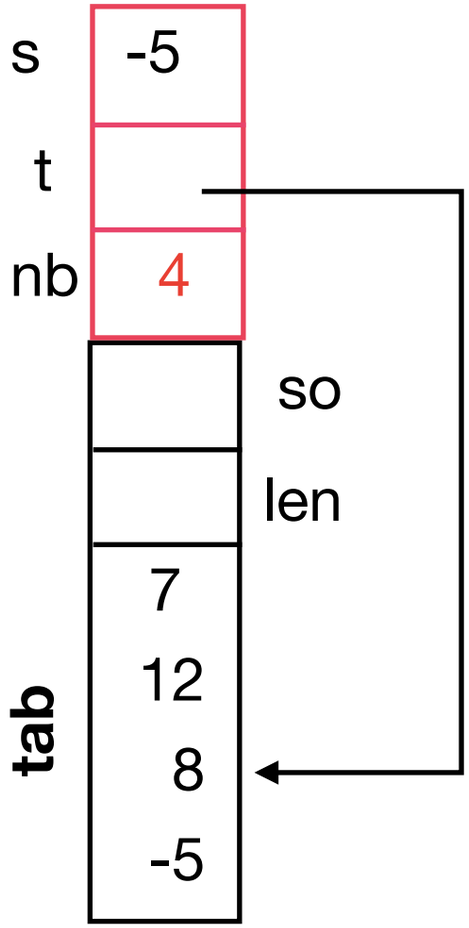
dans la fonction somme :
`sizeof( t ) == sizeof( int * )`

```
int main(void){
  int tab[] = {-5,8,12,7};
  size_t len = sizeof(tab);
  int so = somme( len/sizeof(tab[0]), tab);
}
```

dans main() : `sizeof(tab)` - la taille de tableau tab en nombre d'octets



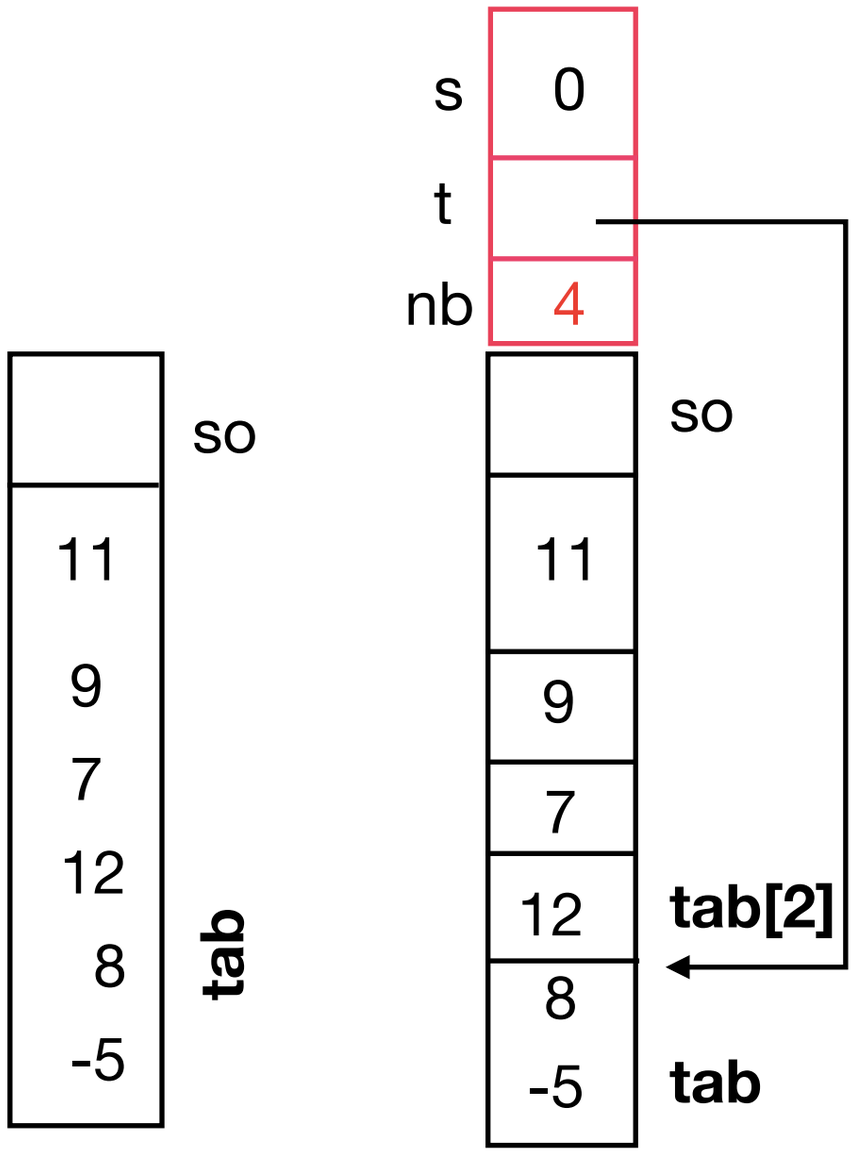

```
s = *t ;
t++;
```



réduction de tableau vers pointeur lors de l'appel de fonction

```
int somme(int nb, int t[]){
  int s = 0;
  for( int i = 0; i < nb; i++){
    s += *t;
    t++;
  }
  return s;
}
```

```
int main(void){
  int tab[] = {-5,8,12,7, 9, 11};
  int so = somme( 3, &tab[2]);
}
```



so <- tab[2]+tab[3]+tab[4]

la fonction **somme** peut prendre comme argument un pointeur

Digression

Il existe aussi les vrais pointeurs de tableau ;

```
int tab[]= {2,-5,12,8};

int *p_t = &tab[0];

int *p_q = tab;

int (*p_tab)[4] = &tab;
```

p_tab est une variable de type "pointeur vers un tableau de 4 int", dont l'utilité est limitée.

Le type de la variable p_tab est différent des types des variables p_t et p_q (mais les trois variables contiennent la même adresse après les trois affectations).

opérateur sizeof

```
sizeof(int)    sizeof(int *)

int a; double b;

  sizeof a/b     -> nombre d'octets
                    pour le type de résultat de a/b
```

`sizeof` s'applique à un type de données ou une expression. Dans le deux cas sizeof donne le nombre d'octets de mémoire nécéssaires pour stocker la donnée.

Le résultat de sizeof est de type

`size_t`

un type entier sans signe, utilisé souvent pour le nombre d'éléments (par exemple le nombre d'élément de tableau)
définie dans `stdlib.h stddef.h` et dans d'autres

Questions

```
int *somme( int n, int tab[]){
    int k;
    for( int i = 0; i < n; i++ ){
       k += tab[i];
    }
    return &k;
}
int main(void){
    int t[]={ 8, 9, 12, -15, -8};
    int *s = somme( 5, t );
    printf("somme = %d\n", *s );
    return 0;
}
```

Est-ce que ce programme est correct?

Qu'est-ce qui se passe avec k quand on fait return de la fonction somme() ?

Réponse

```
int *somme( int n, int tab[]){
    int k;
    for( int i = 0; i < n; i++ ){
       k += tab[i];
    }
    return &k;
}
int main(void){
    int t[]={ 8, 9, 12, -15, -8};
    int *s = somme( 5, t );
    printf("somme = %d\n", *s );
    return 0;
}
```

Qu'est-ce qui se passe avec k quand on fait return de la fonction somme() ?

Après return de la fonction somme(), les variables locales n, tab, k de la fonction somme() sont enlevées de la pile donc somme() retourne l'adresse qui n'est plus valable.

Ce programme n'est pas correct.

Pointeurs pour récupérer plusieurs valeurs de sortie d'une fonction

```
void fun( int n , int tab[],  int *i, int *j){

    /*  trouver les valeurs min et max dans tab[] */

    /*  une fois min et max  trouvés on fait : */

    *i = min;
    *j = max;
    return;
}
int main(void){
    int t[] = {-9, 23, 76, -54, 11};
    int a,b;
    fun(  5, tab, &a, &b );  /* on passe dans la fonction
                              * les adresses de a et b */

}
```

allocation dynamique de la mémoire

allouer et libérer la mémoire

```
#include <stdlib.h>

void *malloc(size_t size)

void *calloc(size_t count, size_t size)

void *realloc(void *ptr, size_t size)  /* à faire plus tard */

void free(void *ptr)
```

`size_t` – type entier non-signé, utilisé souvent pour représenter la taille de données

void * -- pointeur générique, pour l'instant la seule chose qui compte c'est qu'on peut facilement passer de pointeur générique vers un autre pointeur

void *malloc(size_t size)

`malloc(size)` alloue **`size d'octets`** de la mémoire et retourne l'adresse du premier octet de la mémoire allouée.

En cas d'échec `malloc()` retourne `NULL`. **`Vérifiez toujours si malloc() return NULL!`**

Allouer la mémoire pour 20 nombres doubles:

```
double *tab=malloc(20 * sizeof(double));

if(tab == NULL){

   perror("malloc"); exit(1);
}
```

`perror(char *)` - affiche message d'erreur si une fonction (ici malloc()) termine avec erreur

`exit(int)` – termine l'exécution du programme

```
/* initialiser la mémoire allouée*/

for(int i = 0; i < 20; i++){

    tab[i] = i+5; /* même chose que  *(tab+i)=i+5;  */

}
```

void *calloc(size_t nb_elem, size_t elsize)

`calloc()` alloue un tableau de nb_elem éléments, chaque élément de taille elsize d'octets. De plus `calloc()` met à 0 tous les bits de la mémoire allouée. `calloc()` retourne l'adresse du premier octet de la mémoire allouée ou `NULL` en cas d'échec.

```
double *tab = calloc(100, sizeof(double));

/* tab - pointe vers la mémoire pour
 * stocker 100 éléments double.
 * Toutes les valeurs initialisées à 0
*/
```

void free(void *ptr)

`free()` libère la mémoire allouée par `malloc(), calloc()` ou `realloc()`. Le paramètre de `free()` doit être le pointeur retournée par une de ces trois fonctions.

Après l'appel à

```
free( pointeur  )
```

les adresses dans le bloc de la mémoire libérée deviennent invalides.

dangers de malloc()

L'implémentation de malloc() utilise une liste chaînée de blocs alloués.

La taille réelle d'un bloc peut être plus grande que la taille demandée.

Le bloc peut stocker en plus :

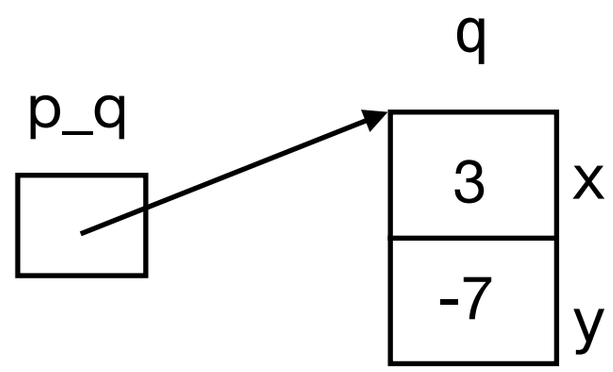
- le pointeur vers le block suivant,
- la taille du block (c'est grâce à cette information que free() "sait" combien de mémoire il doit libérer).

Dangers :

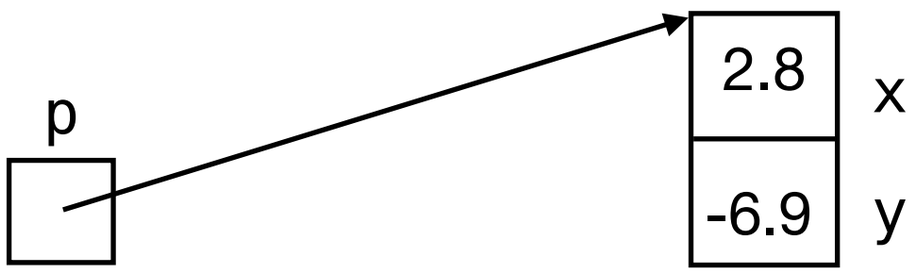
- "memory corruption" : l'écriture dans la mémoire dans le tas au delà des adresses autorisées peut détruire les structures de données de malloc(). La conséquence : malloc(), free() suivants, et les accès à la mémoire, peuvent avoir le comportement imprévisible,
- "memory leak" (fuite de mémoire) : si on "oublie" l'adresse de la mémoire allouée par malloc() il n'est plus possible de libérer cette mémoire. La mémoire allouée s'accumule et nuit à l'exécution du programme. Particulièrement néfaste pour les serveurs qui tournent en permanence.

pointeur de structure

```
typedef struct {
   double x;
   double y;
}point;


point q = { .x = 3, .y =-7 };

point *p_q = &q;

point *p;

p = malloc( sizeof(point) );
```




```
p->x = 2.8;  /* équivalent à  (*p).x = 2.8 */
p->y = -6.9;

p_q->x = p->x + p->y;
```



pointeur de structure - résumé

si p un pointeur de structure alors

p -> champ

donne la valeur de champs de la structure à l'adresse p

C'est équivalent à

(*p).champ

Cette dernière notation ne doit pas être utilisée dans les programmes, la notation avec -> est la notation préférée.