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 */

---

*pointeur dans une expression à droite de = retourne la valeur qui se trouvent à l'adresse pointeur :

`int b = *p_a + 7;` b prendra la valeur 20 (parce que *p_a == 13)

`*pointeur` à gauche de =

`*pointeur = expression;`

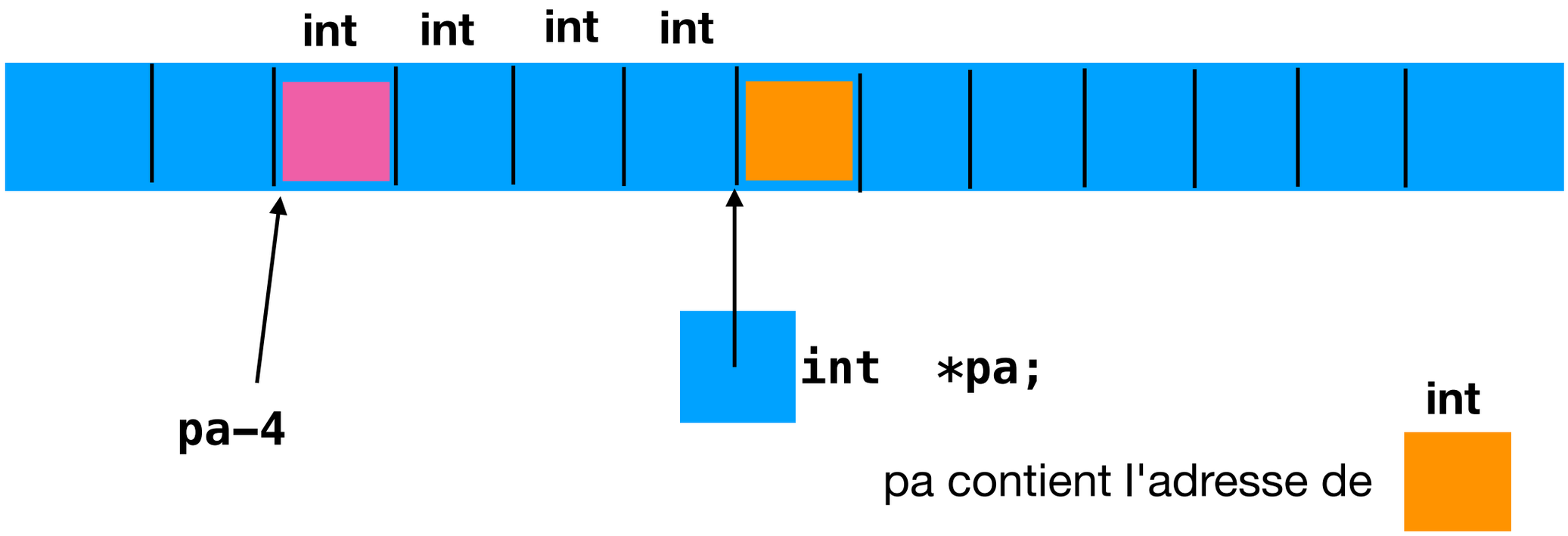
`*p_a = 2 ;` mettre la valeur 2 à l'adresse qui est stockée dans la variable p_a

`*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**



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);
```

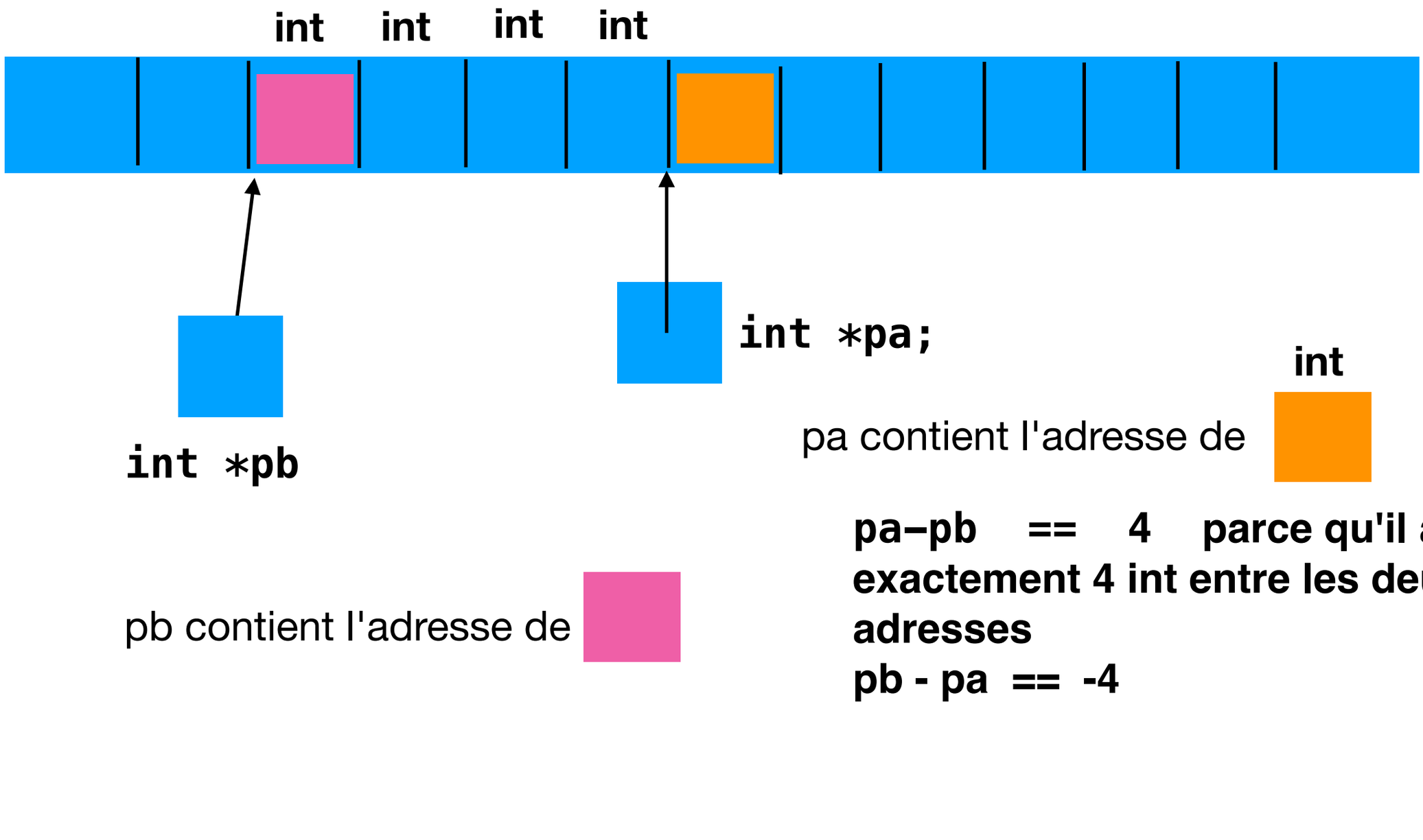
prendre un int qui se trouve à l'adresse **p+1** et un int qui se trouve à l'adresse **p+2**, additionner et mettre le résultat à l'adresse **p-1**

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 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`

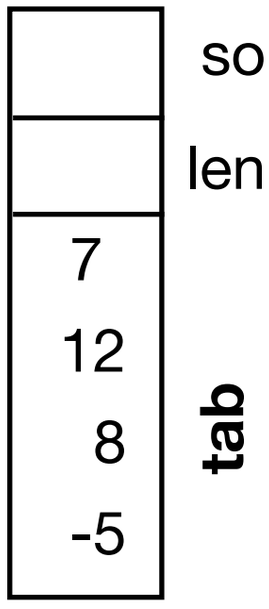
réduction de tableau vers pointeur lors de l'appel de fonction

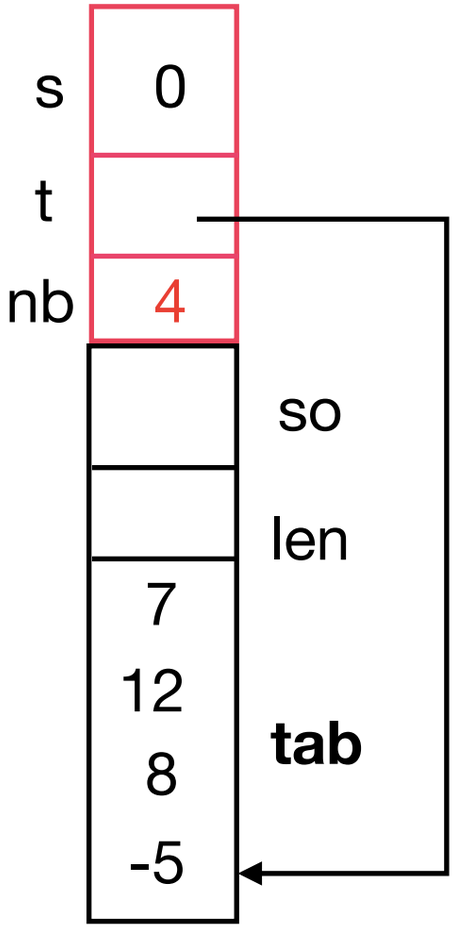
```
int somme(int nb, int t[]){
  int s = 0;
  for( int i = 0; i < nb; i++){
    s += *t;
    t++;
  }
  return s;
}
```

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

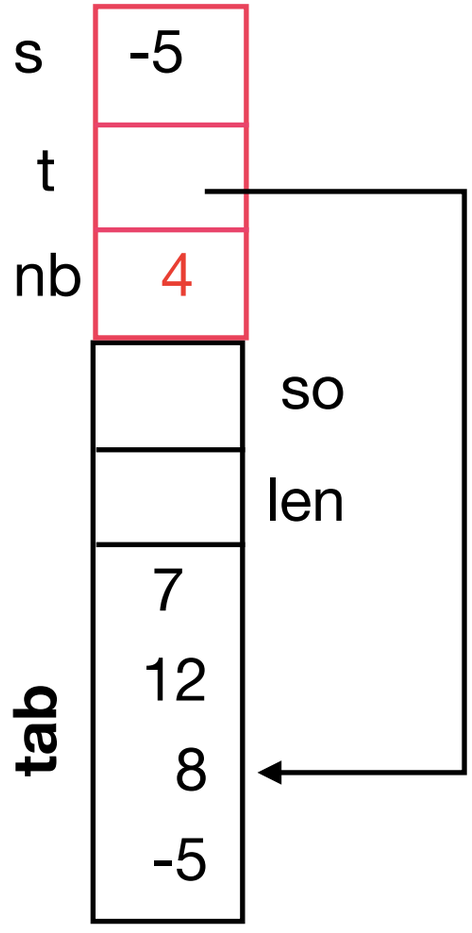
```
int main(void){
  int tab[] = {-5,8,12,7};
  size_t len = sizeof(tab);
  int so = somme( len/sizeof(tab[0]), tab);
}
```

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






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



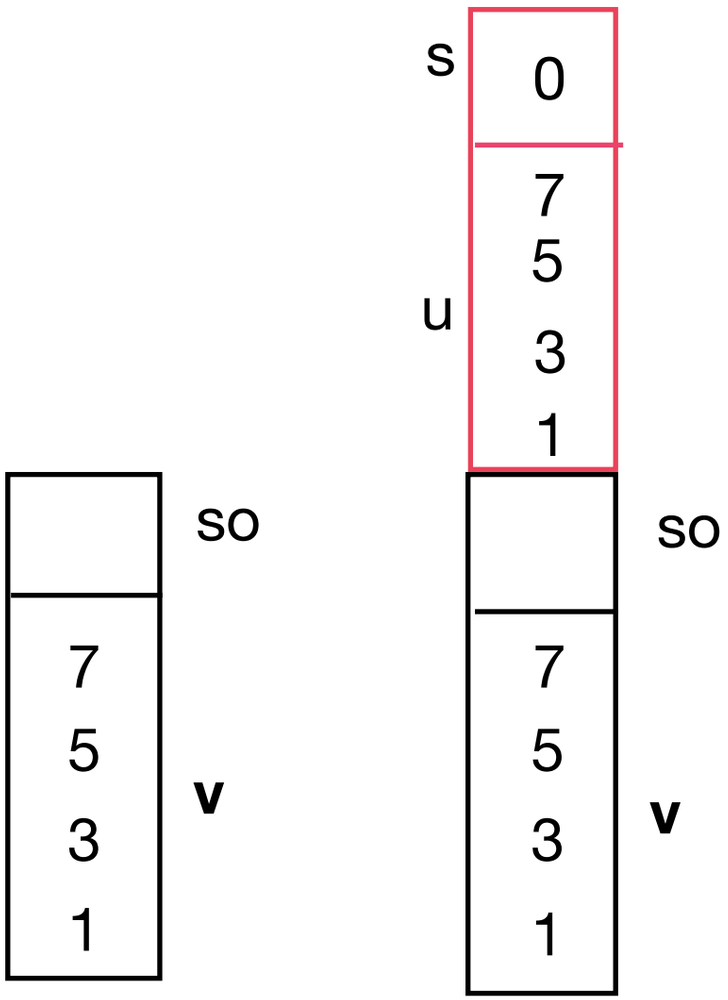
et pour les struct ?

```
int somme( stab u ){
  int s = 0;
  for( int i = 0; i < 4; i++)
    s += u.tab[i];
  return s;
}
```

`sizeof(u.tab)` : la taille de tableau tab en octets

sizeof(u.tab) / sizeof(u.tab[0]) : nb elem

```
typedef struct {
   int tab[4];
}stab;

int main(void){
  stab v = { .tab = {1,3,5,7} };
  int so = somme( v );
}
```



pointeur générique : void *

```
int tab[]={3,4,5,6};

int *p = &tab[1];

void *t=p;  /* p et t contient la même adresse */

char *c=t;
```

Nous pouvons faire une affectation entre un pointeur générique et un autre pointeur sans projection de types, c'est-à-dire sans "cast"). C garantie que la valeur du pointeur est préservée par ces affectations.

A quoi sert le pointeur générique?

Arithmétique de pointeurs ne s'applique pas aux pointeurs génériques:

~~`(t + 1) et (t - 1)`~~

n'ont pas de sens si `t` un pointeur générique (déplacement de combien d'octets ? void n'est pas un type.)

L'application de l'opérateur * n'a pas de sens pour le pointeurs génériques :

~~`int k = *t + 2;`~~

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 nécéssaire pour stocker les données.

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() ?

Quand on fait return de 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.

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)

void free(void *ptr)
```

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

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.`

Allouer un tableau de 20 nombres doubles:

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

if(tab == NULL){
    perror("malloc"); exit(1);
}

for(int i = 0; i < 20; i++){
    tab[i] = i+5; /* même chose que  *(tab+i)=i+5;  */
}
```

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

`exit(int)` **–** termine le processus

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.

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

/* tab - tableau de 100 éléments double
 * initialisés 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.

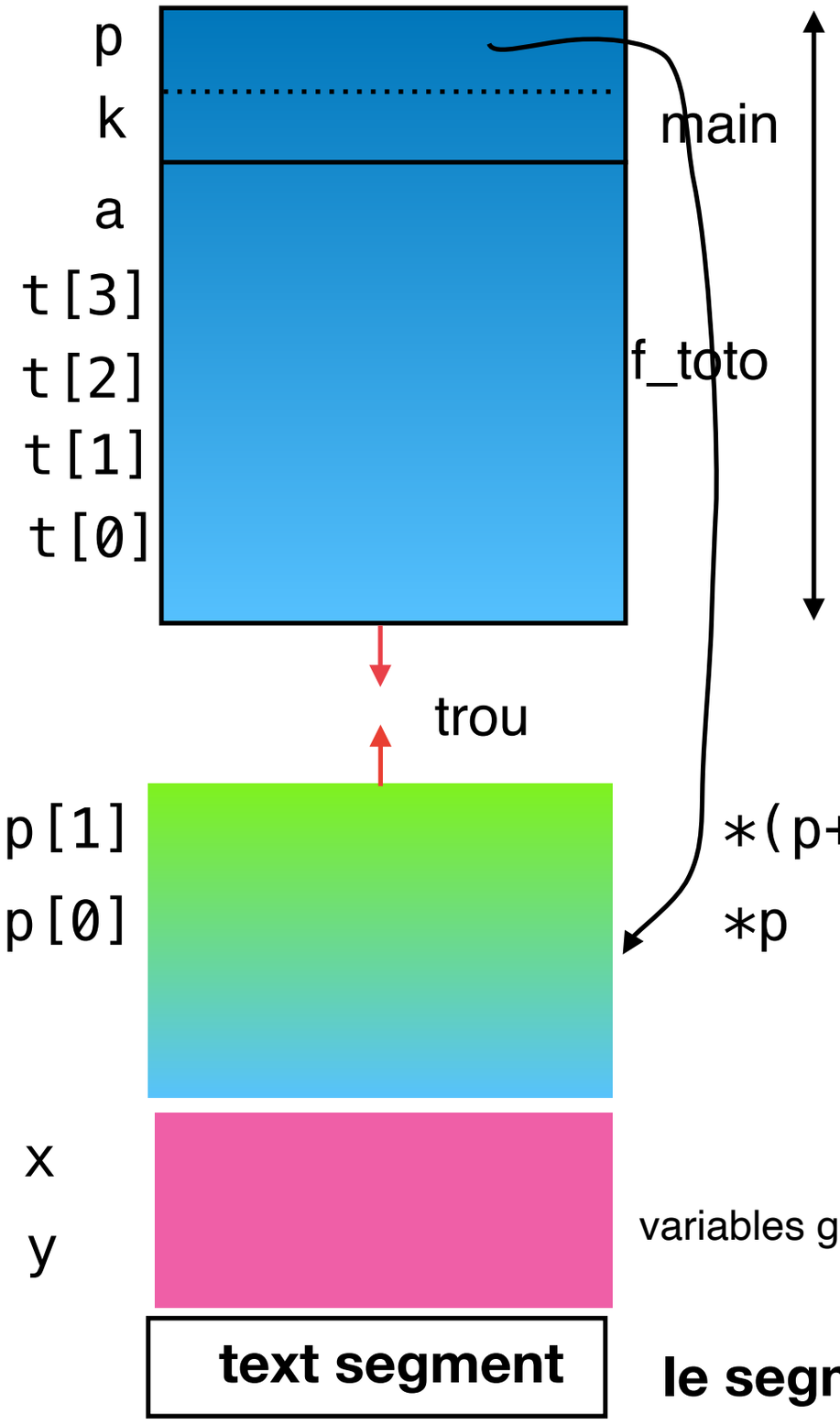
Après l'appel à

`free()`

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

mémoire d'un processus

les adresses mémoire les plus grandes




```c
int x;
int y = 67;

int f_toto(int a){
   int tab[]={1,2,3,4};
   int b;
   ....
}

int main(void){
  int k = 15;
  int *p=malloc(2*sizeof(int));
  p[0]=3; p[1]=33;
  x = f_toto(k + 1);
  .....
}
```

le segment qui contient le code binaire de processus

adresses mémoire les plus petites

mémoire d'un processus

Sur la pile pour chaque appel de fonction:

- les paramètres de la fonction
- les variables locales de la fonction (les variables static déclarées à l'intérieur de la fonction ne sont pas sur la pile)
- l'adresse de retour de la fonction (l'adresse de l'instruction qui sera exécutée juste après return de la fonction)
- peut-être d'autres éléments (par exemple les valeurs de différents registres sauvegardés à l'appel de la fonction, etc.)

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 :

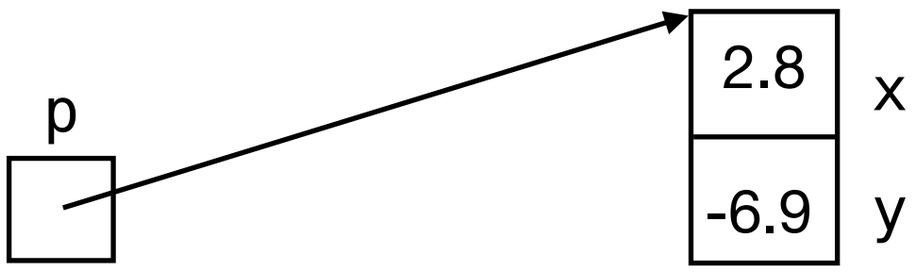
- le pointeur vers le block suivant,
- la taille du block (c'est grâce à cette information que free() "sait" combien de mémoire il doit libérer).

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 = &p;

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;
```



copier ou remplir une zone de mémoire

```
#include <string.h>
```

```
void *memcpy(void *dst, const void *src, size_t n)
```

la fonction copie n octet de l'adresse `src` vers l'adresse `dst`. Si les deux zones chevauchent le résultat est indéfini. La fonction retourne `dst`.

```
void *memmove(void *dst, const void *src, size_t n)
```

même chose que la fonction précédente mais les deux zone peuvent chevaucher.

```
void *memset(void *s, int c, size_t n)
```

la fonction copie la valeur de c (transformée en `unsigned char`) sur n octets à partir de l'adresse s

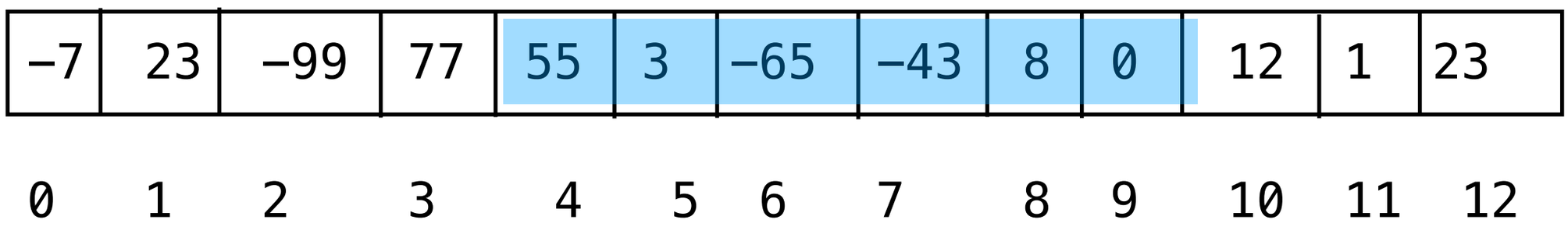
exemple : copier une partie de tableau

```
int tab[]={ -7, 23, -99, 77, 55, 3, -65, -43, 8, 0, 12, 1,
-23} ;
```

```
int *autre_tab = malloc( sizeof(int) * 6);

memmove(autre_tab, &tab[4], 6 * sizeof(int) );
```

copie 6 éléments de `tab` à partir de tab[4] à l'adresse `autre_tab`.

tab



autre_tab