Langage C

Wieslaw Zielonka
zielonka@irif.fr

complements sur les entiers

les valeurs min et max pour les entiers

#include <limits.h>

Le fichier en tête limits.h définit plusieurs constantes symboliques utiles :

valeur max	valeurs min
SCHAR_MAX	SCHAR_MIN
SHRT_MAX	SHRT_MIN
INT_MAX	INT_MIN
LONG_MAX	LONG_MIN
LLONG_MAX	LLONG_MIN

UCHAR_MAX
UINT_MAX -- unsigned int maximal
ULONG_MAX -- unsigned long maximal

etc.

#include <stdint.h>

Le fichier en tête stdint.h définit plusieurs types entiers dont le nombre de bits est fixe et indépendant de l'architecture :

int8_t entier sur 8 bit

int16_t

int32_t entier sur 32 bits $[-2^{31}, 2^{31}-1]$

uint8_t

uint16_t

uint32_t entier sans signe de 32 bits $[0, 2^{32}-1]$

int64_t et uint64_t peuvent être définis (mais ce n'est pas une obligation si l'architecture ne les supporte pas)

Initialisations de tableaux et structures

Variables et tableaux sans initialisation

```
int i;
int j = 15;
#define SIZE 124
double tab[ SIZE ];


int fun( .... ){
        int l = 128;
        int k;
        int t[SIZE] ;
}
```

Les variables et tableaux globales (niveau 0) sans initialisation sont initialisées avec **0.**
`i` **et** `tab[]` seront initialisé avec zéro

`k` et `t[]` définis à l'intérieur d'une fonction ne sont pas initialisés et contiennent les valeurs indéterminées.

Initialisation de tableaux

```
#define SIZE 124

int t[ SIZE ] = { 1, -2, 4 };
/*  t[0]==1, t[1]==-2, t[2] = -4,
 *  t[i]==0 pour i de 3 à 123 */

int d[ SIZE ] = { [4]=11, [8]=22, [20]=33 };
/*  d[4]==11, d[8]==22, d[20] == 33, tous les autres
éléments initialisés à 0 */

int tab[ SIZE ] = { [4]=11, 22, 33, [20]=-7, 8};
/*  tab[4] == 11, tab[5]==22, tab[6]==33, tab[20]=-7,
    tab[21]=8, tous les autres éléments initialisés à 0*/
```

Initialisation de tableaux/structures

```
#define SIZE 124

struct pol{
    int n;
    double tab[SIZE];
};

struct pol u = { .n = 20 };
/* u.n == 20,  dans u.tab tous les éléments initialisés à 0
les champs de la structure qui ne sont pas initialisés explicitement prennent la valeur
0 */

struct pol tri; /* variable tri définie dans une fonction n'est pas initialisée, les
valeurs de champs sont indéterminées*/


int kt[SIZE] = {} ;
non, la norme du C exige la liste d'initialisation soit non-vide. Mais gcc accepte et
initialise les éléments de kt[] à 0.


La forme correcte pour initialiser tous les éléments de tableau à 0 :
int kt[SIZE] = { 0 };


int ht[SIZE] = { 5 }; /* ht[0]==5, tous les autres à 0 */
```

Initialisation de tableaux/structures

On ne peut pas initialiser les tableaux de longueur variable:

```
int fun( int n ){

    int tab[ 2*n+10] = { 1,2,3 };  ne compile pas

    int k = 5;
    double T[ k ] = {3, 5, 7, 9, 6}; ne compile pas
    ......
}
```

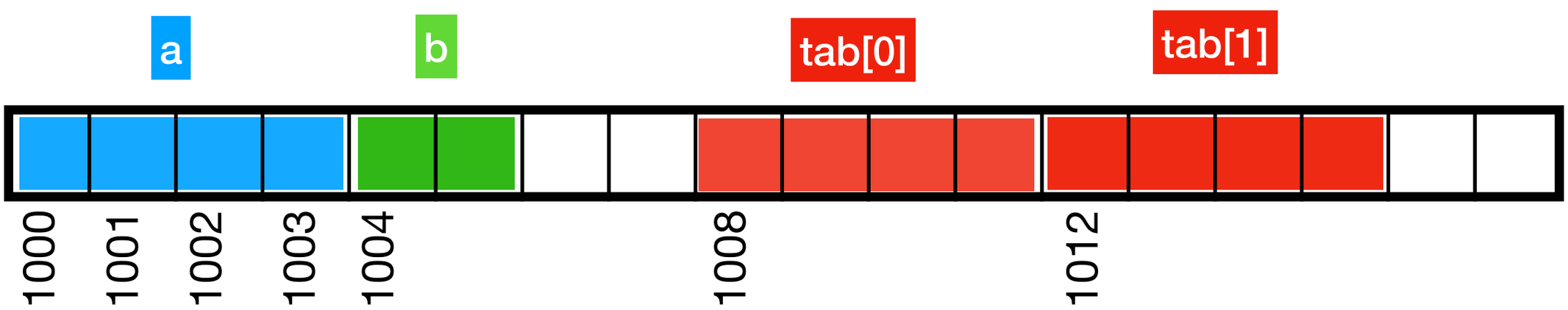
initialement les valeurs dans des tableaux de longueur variables sont toujours indéterminées

Pointeurs - arithmétique de pointeurs

Pointeurs == les adresses

```
int a; short b;

int tab[] = {1,-12};

a = -6 ; b = 7;
```

On assume ici que sizeof(int) == 4
et sizeof(short) == 2



Chaque octet de la mémoire possède une adresse unique. L'ordre de données dans la mémoire pas forcement le même que l'ordre de déclaration.

On peut avoir des "trous" dans la mémoire, ce sont des octets qui ne sont pas utilisés pour stocker les données.

Pourquoi les trous?

Alignement : par exemple l'adresse d'un int doit être un multiple de sizeof(int).

Pointeurs == les adresses

Dans les machines modernes les adresses comportent 64 bits. Pour afficher l'adresse comme un nombre on utilise la notation hexadécimal.

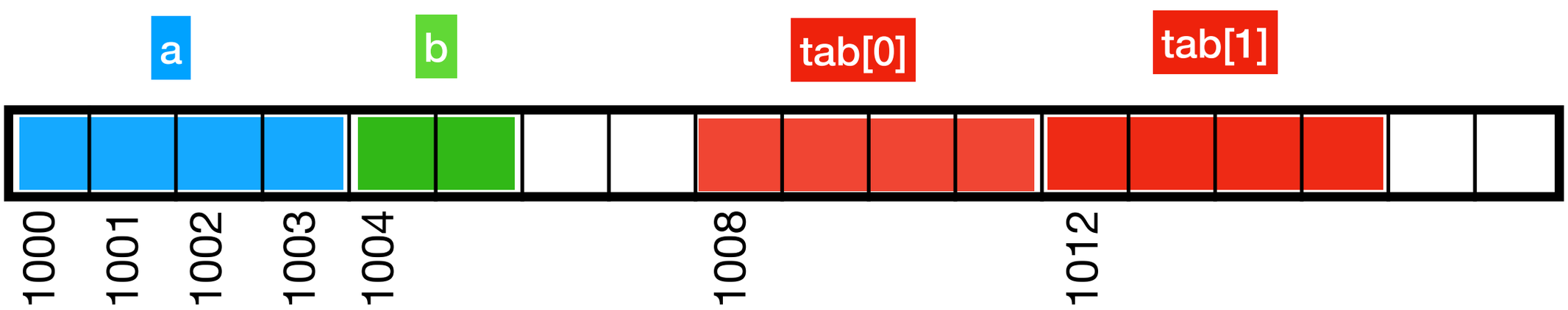
héxa	décimal	binaire	héxa	décimal	binaire
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	A	10	1010
3	3	0011	B	11	1011
4	4	0100	C	12	1100
5	5	0101	D	13	1101
6	6	0110	E	14	1110
7	7	0111	F	15	1111

Donc l'adresse de 64 bits aura 16 chiffres héxa, par exemple 0x00007ffeefbff728 et l'adresse de 4 octets plus loin est 0x00007ffeefbff73C

Pointeurs : les adresses
Opérateur &

```
int a; short b;

int tab[] = {1,-12};

a = -6 ; b = 7;
```

On assume ici que sizeof(int) == 4
et sizeof(short) == 2



Chaque octet possède une adresse unique.

&a -> l'adresse (du premier octet) de a

&b -> l'adresse (du premier octet) de b

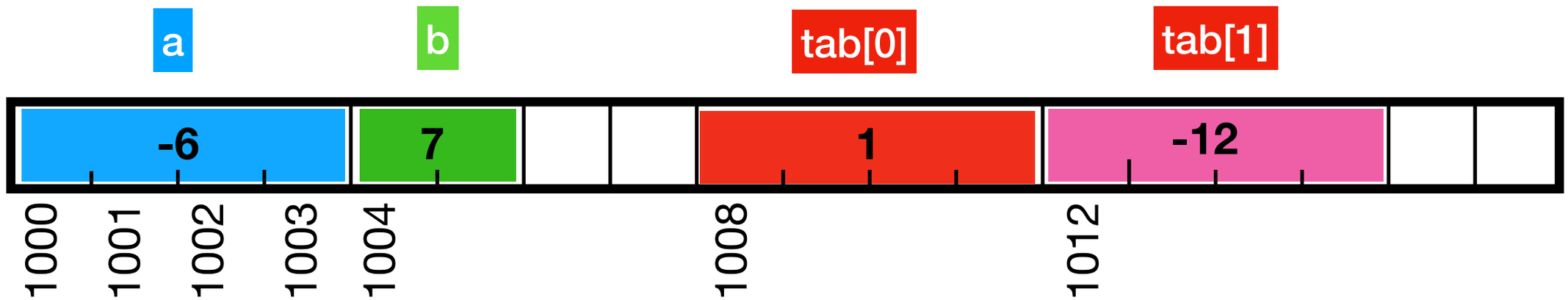
&tab[0] -> l'adresse (du premier octet) de tab[0]

&tab[1] -> l'adresse (du premier octet) de tab[1]

tab == &tab[0] -> l'adresse (du premier octet) de tab

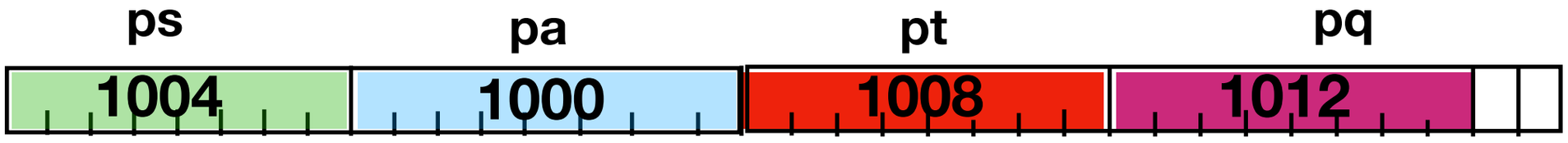
Pointeurs : les adresses
Opérateur &

```
int a; short b; int tab[] = {1,-12};

a = -6 ; b = 7;
```



les variables de type pointeur pour mémoriser les adresses:

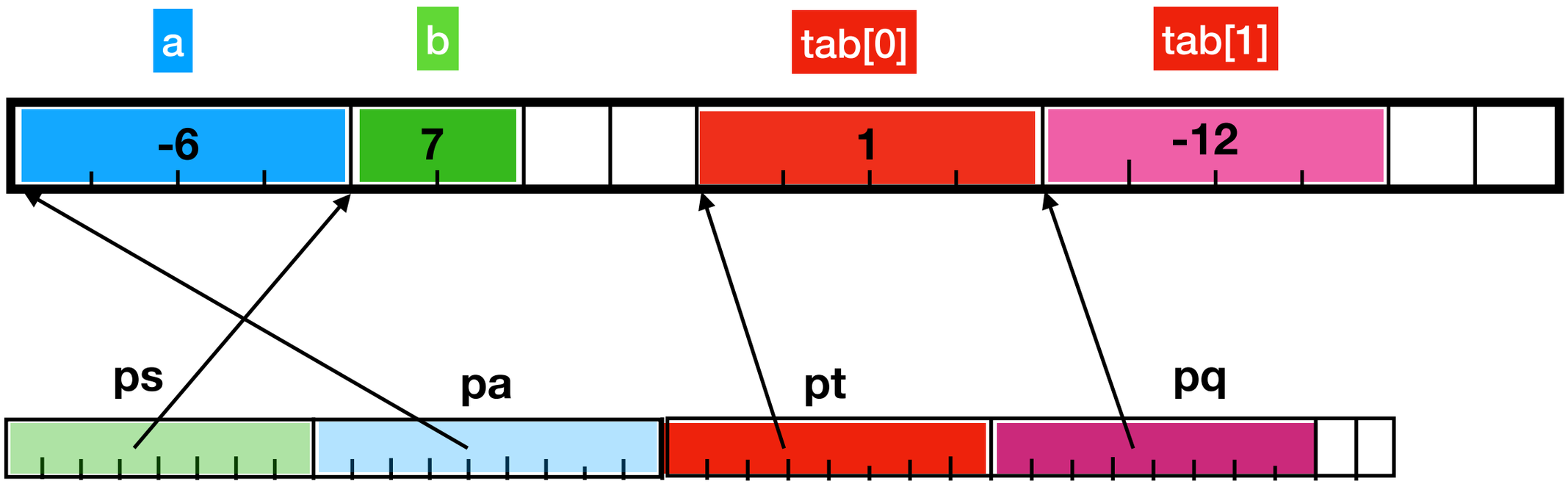
```
short *ps = &b;          int *pa = &a;
int   *pt = &tab[0];     int *pq = &tab[1];
```



Pointeurs : les adresses
Opérateur &

```
int a; short b;

int tab[] = {1,-12};

a = -6 ; b = 7;
```

```
short *ps = &b;        int *pa = &a;
int   *pt = &tab[0];    int *pq = &tab[1];

short *   pointeur vers une short
int *     pointeur vers un int
```



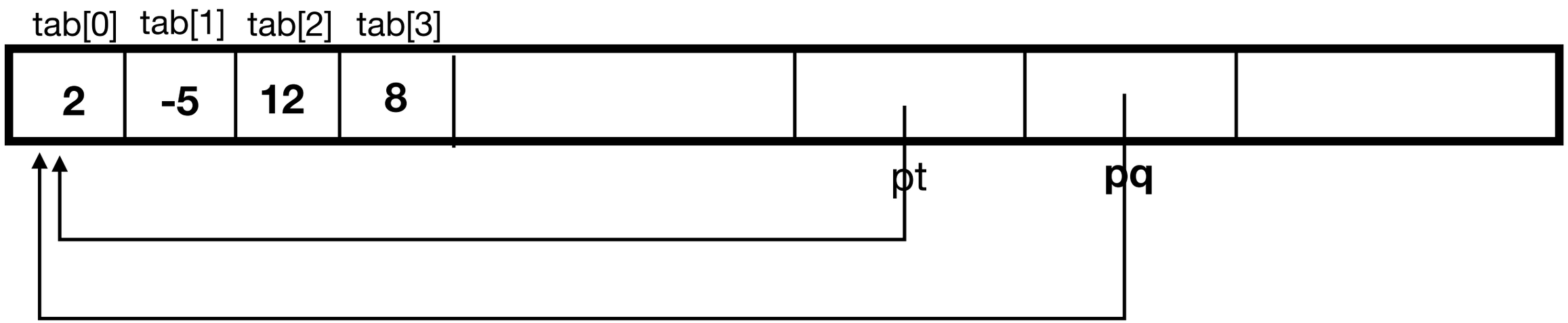
Attention : l'ordre réel de variables dans la mémoire peu être différent

Pointeurs et tableaux

En C le nom de tableau dans une expression est évalué comme l'adresse du premier élément du tableau.

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

int *pt = &tab[0];

int *pq = tab;  /* sans & devant le nom
                * du tableau*/
```

Les variables pt et pq contiennent l'adresse du premier élément de tab.



Déclarer plusieurs pointeurs d'un coup

Attention à la notation :

```
int *a, *b;
```

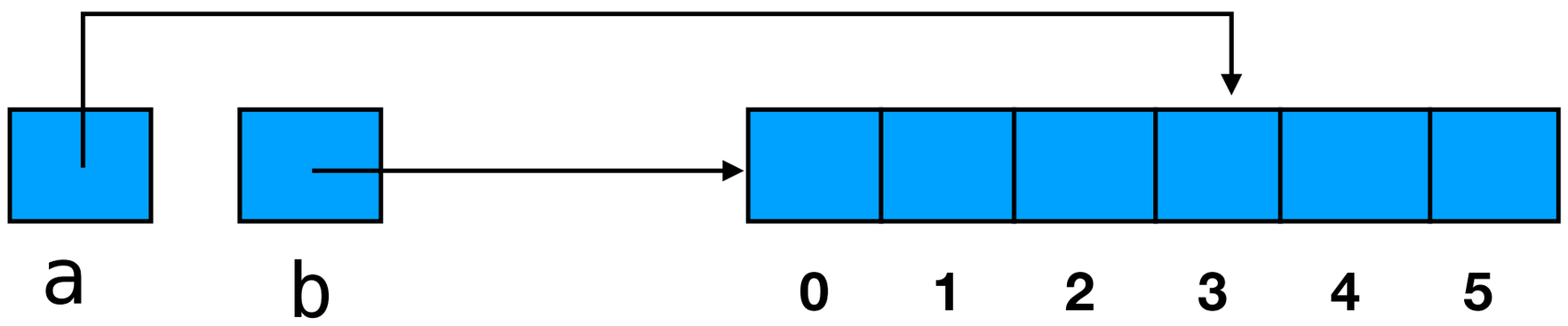
Deux variables pointeurs ***a, b*** de type ***int*** * déclarées d'un seul coup, différent de

```
int *c, d;
```

c est un pointeur vers int,

d une variable int, pas un pointeur.

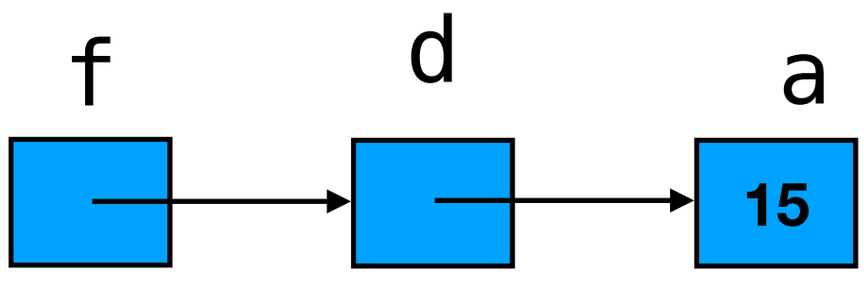
```
int tab[6];  a = &tab[3]; b = &tab[0];
```



pointeur de pointeur

```
int a = 15;

int *d;   /* d un pointeur vers un int   */

int **f;   /* f un pointeur vers un "int *", autrement

              f sert à stocker l'adresse d'une donnée de

              type "int *"                    */
```

```
d = &a;    f = &d;
```



opérateur * appliqué au pointeur à gauche de l'affectation

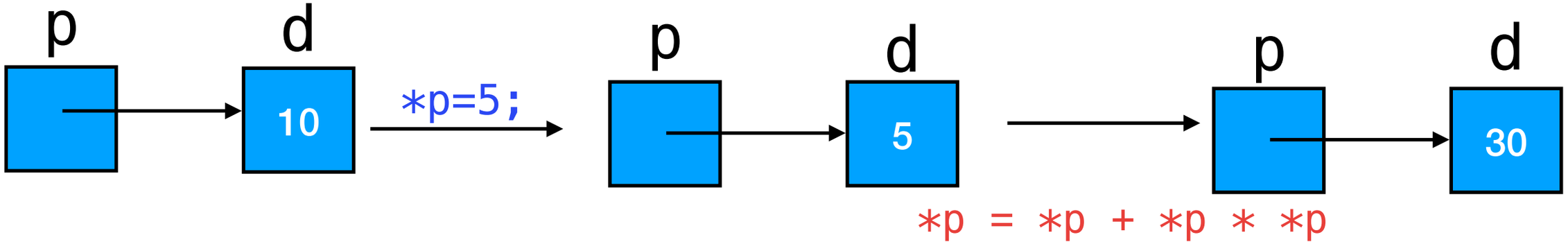
```
int *a;

int d = 8;

a = &d;
```


```
*a = 12;   /* mettre la valeur 12 à l'adresse stockée
            * dans a */

printf("%d\n", d); /* affiche 12 */
```

a de type int *
***a de type int**

opérateur * dans une expression pour récupérer les données

```
int *p;  int d; d = 10;
p = &d;
*p = 5;  /* mettre  la valeur 5 à l'adresse stockée dans p */
printf( "d=%d\n", d);   -->  d=5  d change la valeur
*p = *p + *p * *p ;
printf( "d=%d\n", d);            -->  x=30    x change la valeur
```



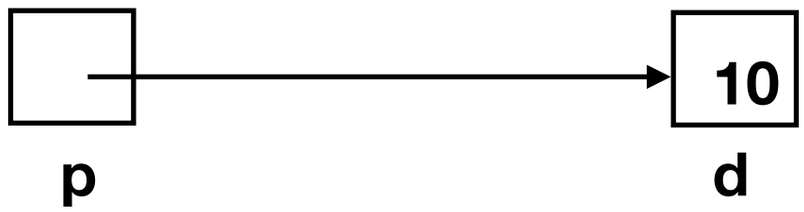
*p dans une expression c'est la valeur stockée à l'adresse p

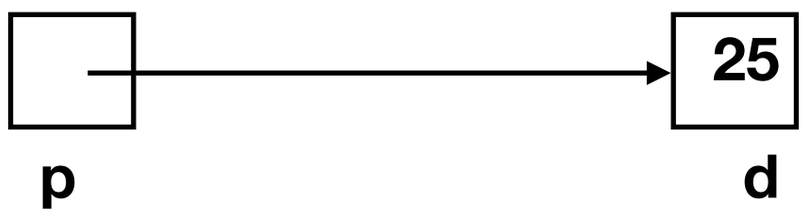
***p + *p * *p**

*p == 5   donc *p + *p * *p == 5 + 5*5 == 30

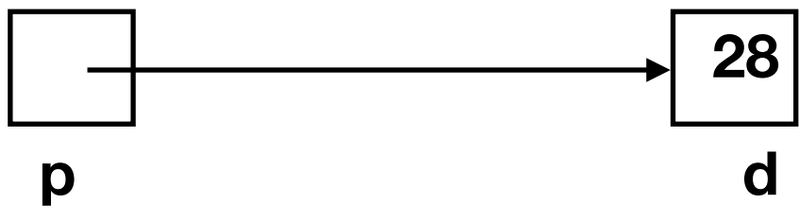
opérateur * appliqué à un pointeur

```
int d;

int *p = &d;

d = 10;

*p = (*p) * 2 + 5;        /* d prend la valeur 2*10 + 5 = 25 */
```

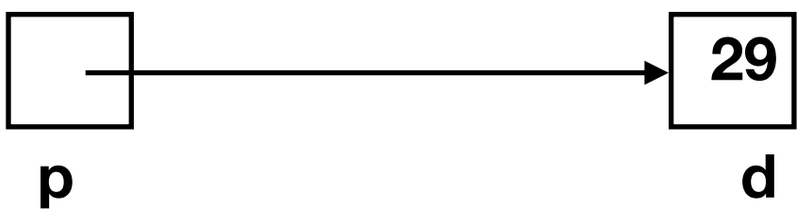





```
*p += 3;      /* incrémenter de 3 la valeur stockée à l'adresse p;
               * d reçoit 28 */
```

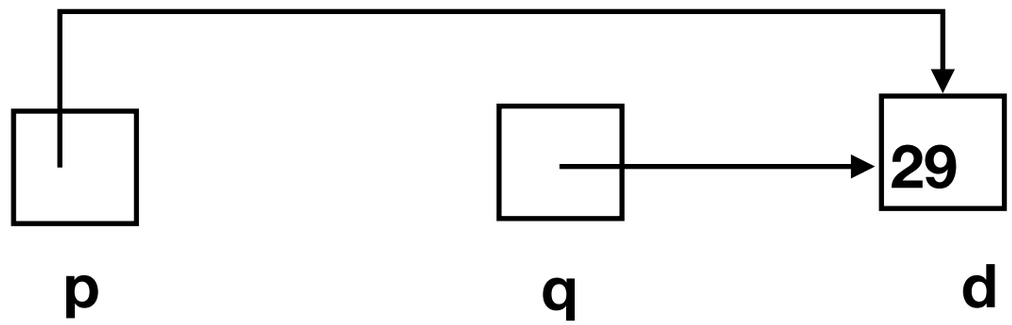


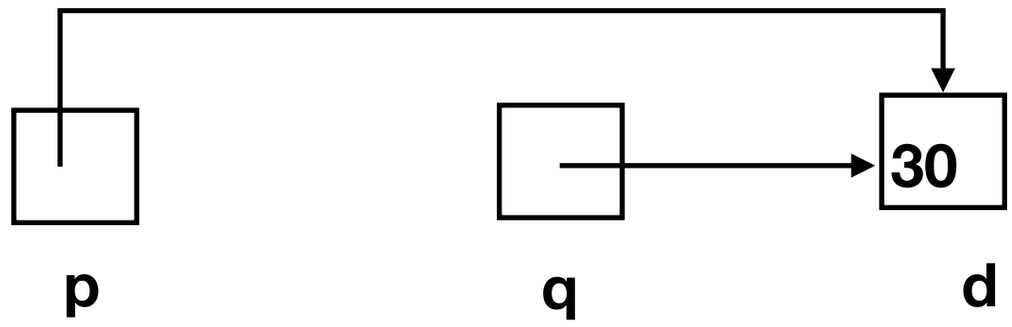

```
++(*p) ; /*   incrémenter un int qui se trouve à l'adresse donnée
          *   par p, d == 29   ++ s'applique à la valeur qui se trouve
          *   à l'adresse p     */
```



opérateur * appliqué à un pointeur

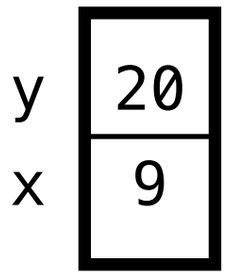
```
int d;

int *p = &d;

int *q = p ;  /* les pointeurs p et q contiennent
l'adresse de d */

(*q)++; /* (*q)++ augmente la valeur int à l'adresse q,
*  d == 30 */
```

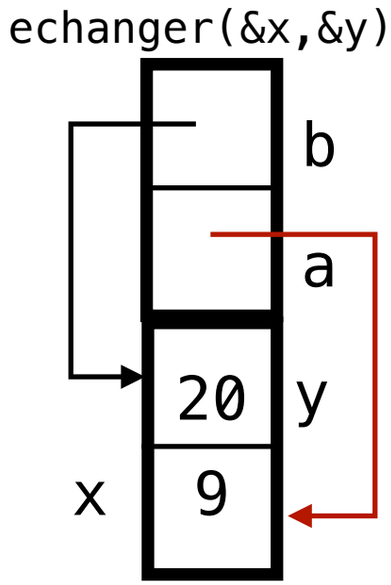




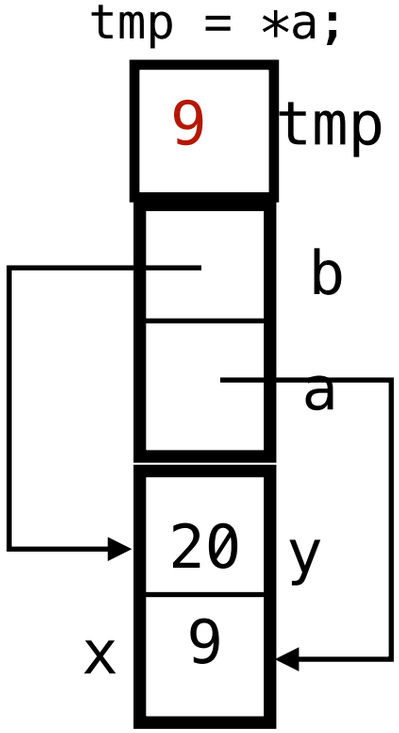
les pointeurs et les arguments de fonctions

```
void
echanger( int *a, int *b)
  {
    int tmp = *a;
    *a = *b;
    *b = temp;
    return;
}


int main(void){
    int x=9, y=20;
    ......
    echanger( &x, &y);

}
```

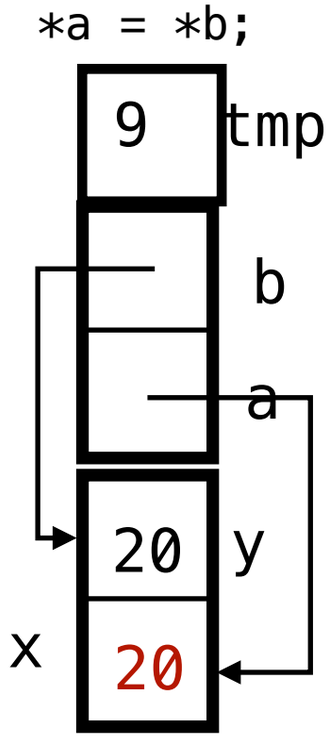




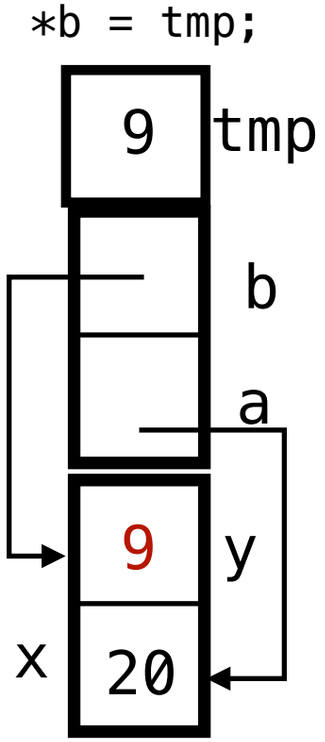
tmp = *a;



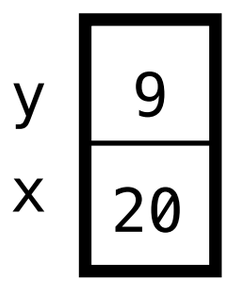
*a = *b;



*b = tmp;



return



pointeur NULL

NULL défini dans : stdio.h   stddef.h

```
int *pi = NULL;

double *pd = NULL;
```

NULL une valeur spéciale pour les pointeurs, différente de toutes les adresses réelles.

Quand pd == NULL

*pd = 5;

provoque l'envoie d'un signal qui termine l'exécution de programme.

arithmétique de pointeurs

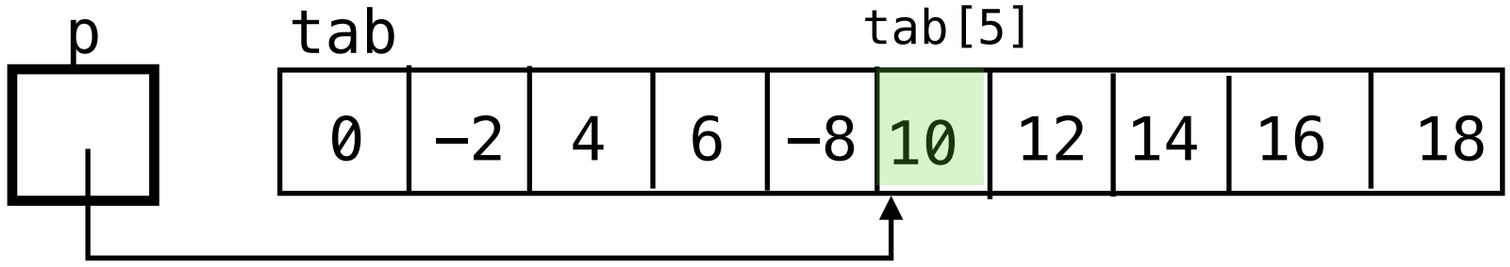
```
int tab[]={0, -2, 4, 6, -8, 10,12,14,16,18};
int *p = &tab[5];
printf("%i \n", *p);
p = p + 3;
printf("%i \n", *p);
p = p - 5;
printf("%i \n", *p);
```

10

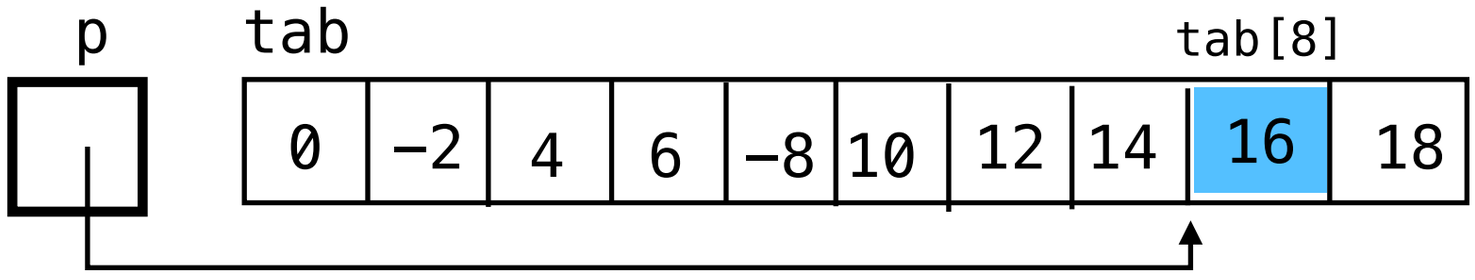
16

6

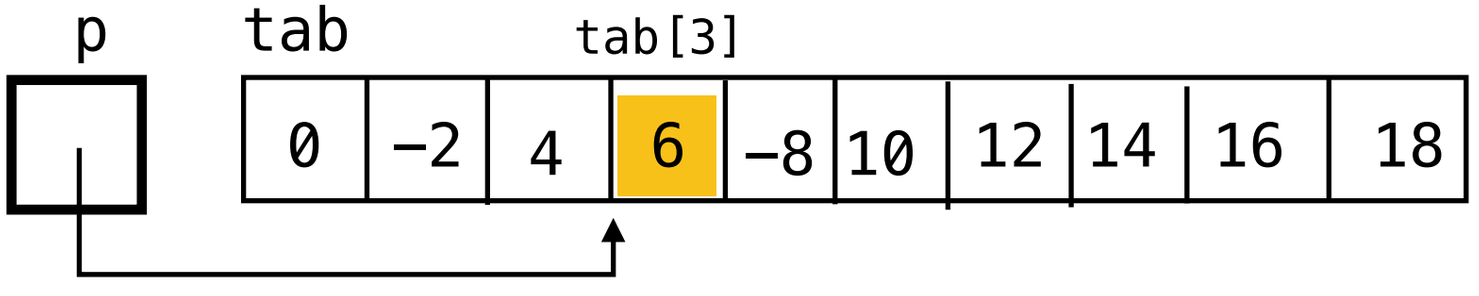
p = &tab[5];



p += 3;



p = p - 5;

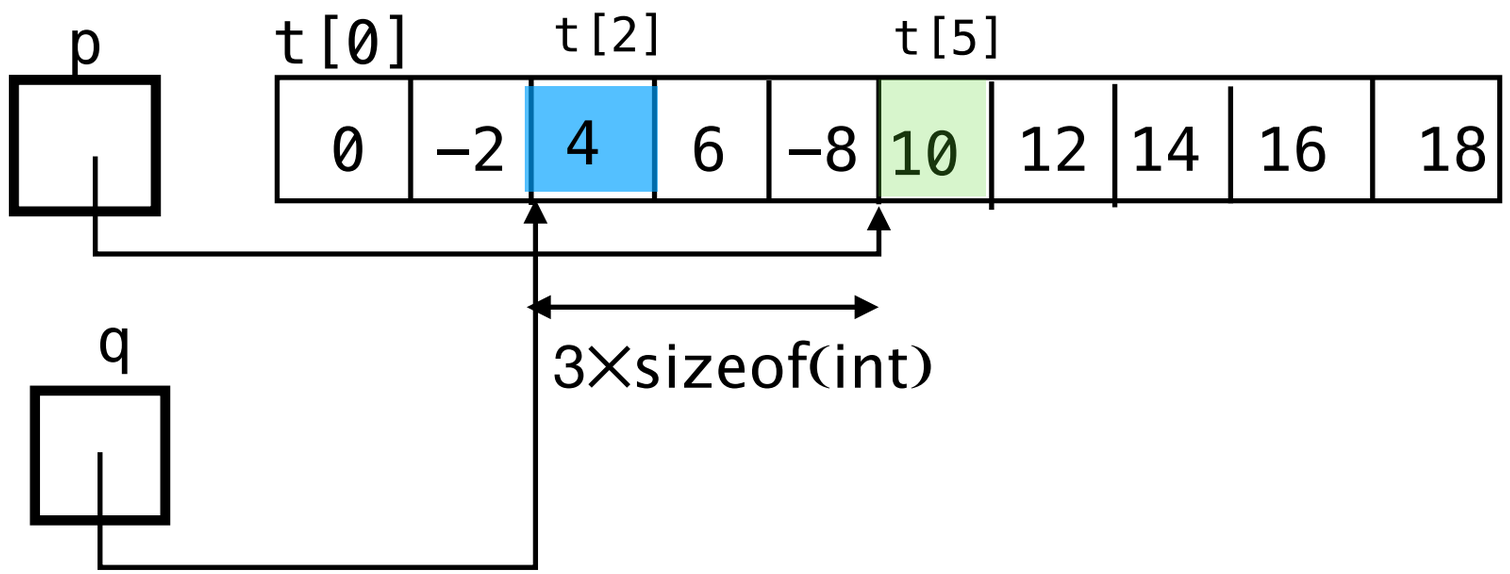


arithmétique de pointeurs

```
int t[]={0,-2,4,6,-8,10,12,14,16,18};
int *p = &t[5];

int *q = p - 3;
```

p = &t[5];

q = p - 3;



Si p est un pointeur vers une donnée de type t :

```
t  *p;
```

et n une expression de type int alors les adresses

```
p + n    et   p - n
```

dépendent de type `t` du pointeur. Le décalage de l'adresse calculé en nombre d'octets est de

```
n * sizeof( t )
```

arithmétique de pointeurs

```
unsigned int tb[] = { 4, 8 };
unsigned int   *q_int;
unsigned char  *q_char;

q_int = &tb[0];
```

```
q_char = &tb[0];  --> warning: incompatible pointer types
les types de pointeurs doivent être les mêmes,
à gauche pointeur vers unsigned char,  à droit pointeur vers unsigned int
```

```
/* prendre l'adresse de tb[0] mais la traiter comme l'adresse de unsigned char
*/
q_char = (unsigned char *) &tb[0];


/*     afficher les deux pointeurs
 *     %p  le format pour pointeur */
printf("q_int == %p, q_char == %p\n", q_int, q_char) ;
```

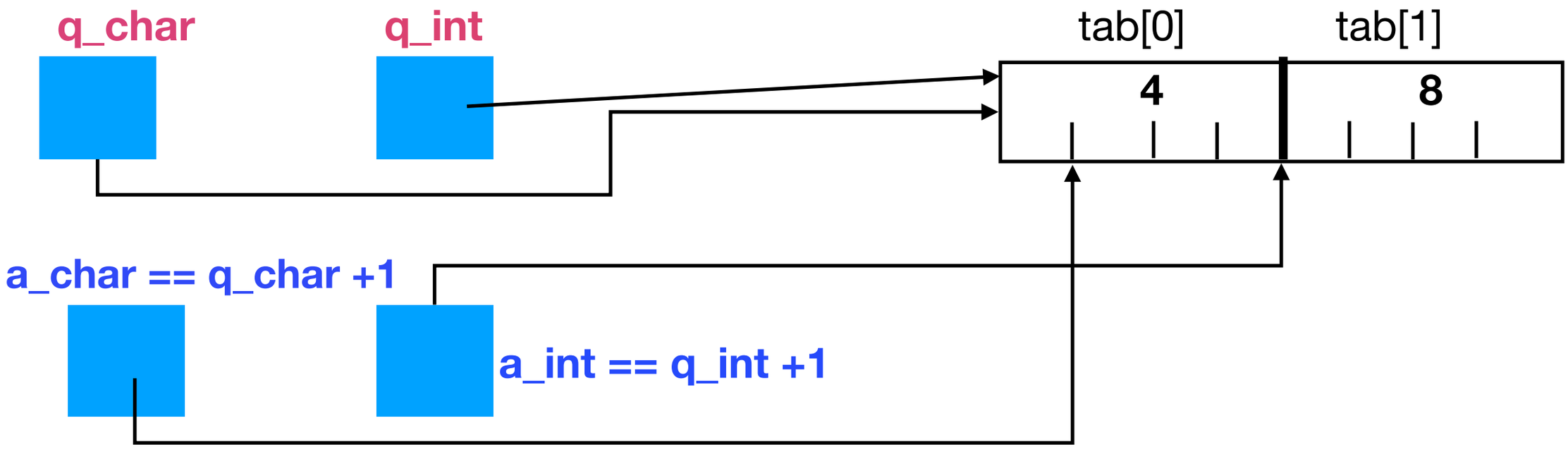
sur mon portable affiche :

```
q_int == 0x7ffee115993c, q_char == 0x7ffee115993c
```

q_int et q_char contiennent exactement la même adresse (affichage de l'adresse en hexadécimal) même si les types de deux pointeurs différents

arithmétique de pointeurs
décalage

```
unsigned int tb[] = { 4, 8 };
unsigned int *q_int, *a_int;
unsigned char *q_char, *a_char;

q_int = &tb[0];
q_char = (unsigned char *) &tb[0];   q_int == 0x7ffee115993c, q_char == 0x7ffee115993c

a_int = q_int +1;  a_char = q_char +1;

printf("a_int == %p,  a_char == %p\n", a_int, a_char );

mon portable affiche :
           a_int  == 0x7ffee1159940,  a_char == 0x7ffee115993d
                     0x7ffee115993c + 4 == 0x7ffee1159940
                     0x7ffee115993c + 1 == 0x7ffee115993d
sizeof(unsigned char) == 1 sizeof(unsigned int)  == 4
*a_int  == 8    *a_char == ???
```

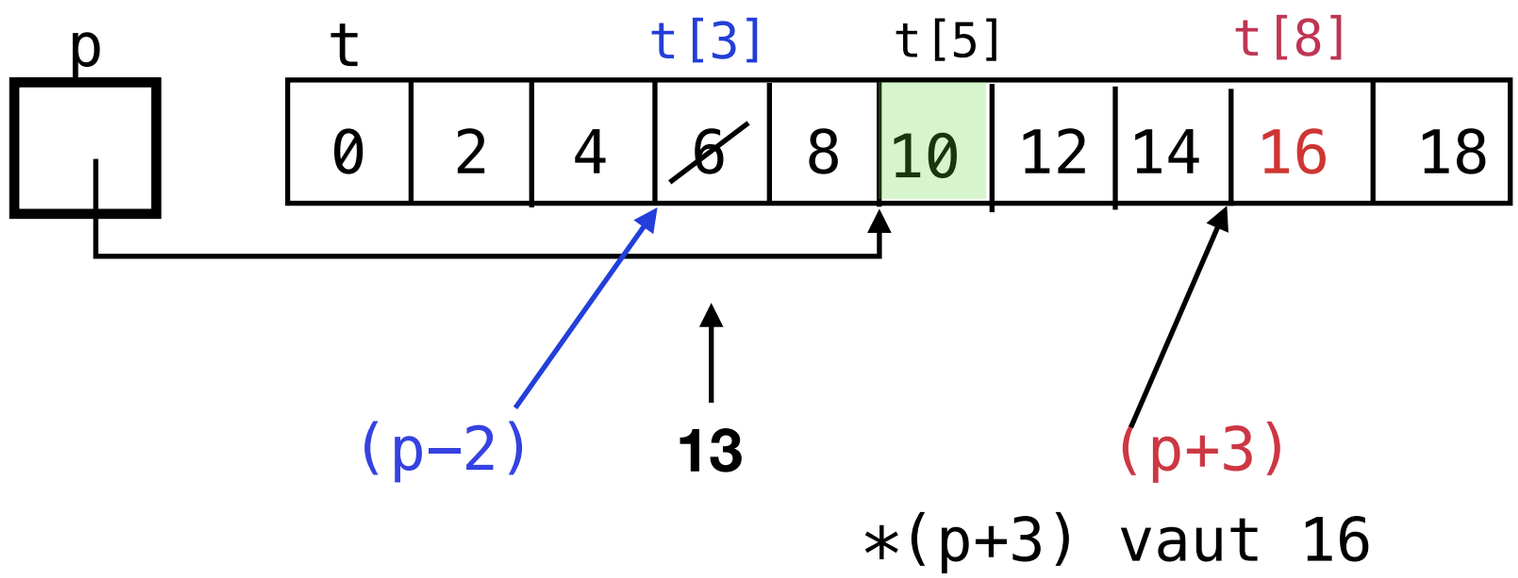


arithmétique de pointeurs

```
int t[]={0,2,4,6,8,10,12,14,16,18};
int *p = &t[5];

*(p - 2) = *(p + 3) - 3;  /* p[-2]=p[3]-3;  */
```

Dans l'expression à droite :
*(p+3) la valeur int stocké à l'adresse (p+3), donc 16.

L'expression à gauche *(p-2) indique qu'il faut mettre la valeur de l'expression à droite à l'adresse (p-2). La valeur de pointeur p n'est pas modifier, c'est la valeur stockée sur les sizeof(int) octet



Nouvelle valeur de t[3] est *(p+3) - 3 = 16 - 3 = 13

arithmétique de pointeurs

```
int *p;

int k;
```

Dans une expression à droite de l'affectation

`*(p-k)` et `*(p+k)`

donnent la valeur de la donnée qui se trouvent à l'adresse `p-k` et `p+k` respectivement.

Le décalage k est mesuré en nombre d'éléments de type `int` (k×sizeof(int) si le décalage compté en nombre d'octets).

En général, k peut être une expression quelconque dont la valeur est un entier.

arithmétique de pointeurs

```
int *p; /* alpha, un type quelconque*/

int k;
```

Le compilateur C traduit

`p[k]` et `p[-k]`

automatiquement en :

`*(p-k)` et `*(p+k)`

Avec les pointeurs nous pouvons utiliser la même notation que avec les tableaux:

```
int t[]={0,2,4,6,8,10,12,14,16,18};
int *p = &t[5];

p[-2] = p[3] - 3;

/* au lieu de   *(p - 2) = *(p + 3) - 3;*/
```

arithmétique de pointeurs - les erreurs

Le compilateur du C ne fait (presque) aucune vérification si les adresses calculées à l'aide de pointeurs sont "correctes".

Exemple: le programme suivant, manifestement erroné, a été compilé et exécuté sur MacOS sans erreur ni warning (mais produit des résultats bizarres).

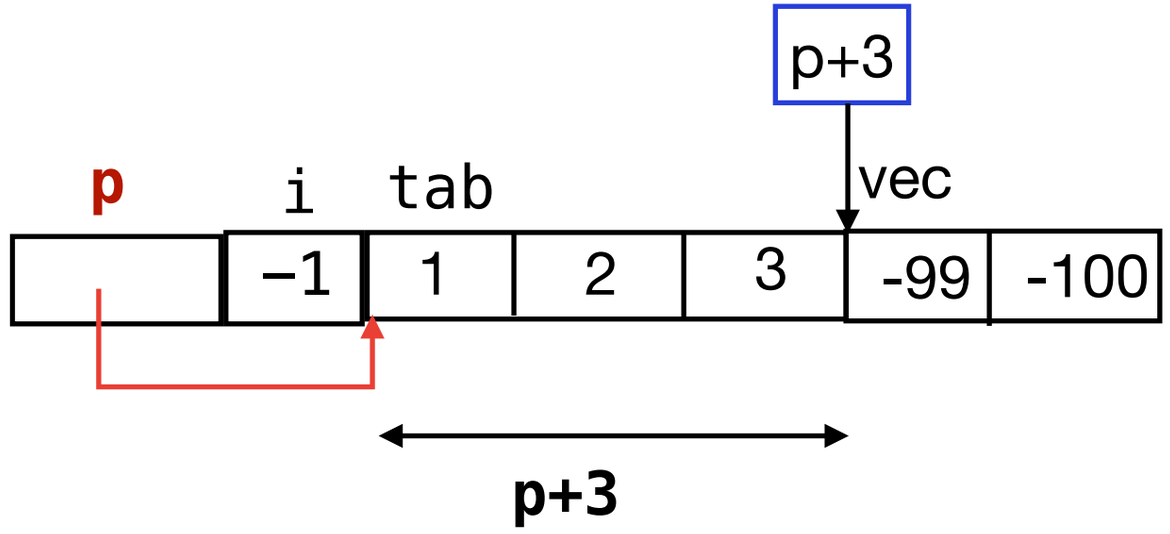
arithmétique de pointeurs

```
int vec[] = {-99, -100};
int tab[] = {1, 2, 3};
int i = -1;
int *p = &tab[0];
 printf("&p = %p \n&i = %p  \n&tab[0] = %p  \n&vec[0] = %p \n",
   &p, &i , &tab[0], &vec[0] );
```

format pour afficher un pointeur

Sur mon MacBook printf affiche :

```
&p = 0x7ffee28c5980
&i = 0x7ffee28c5988
&tab[0] = 0x7ffee28c598c
&vec[0] = 0x7ffee28c5998
```



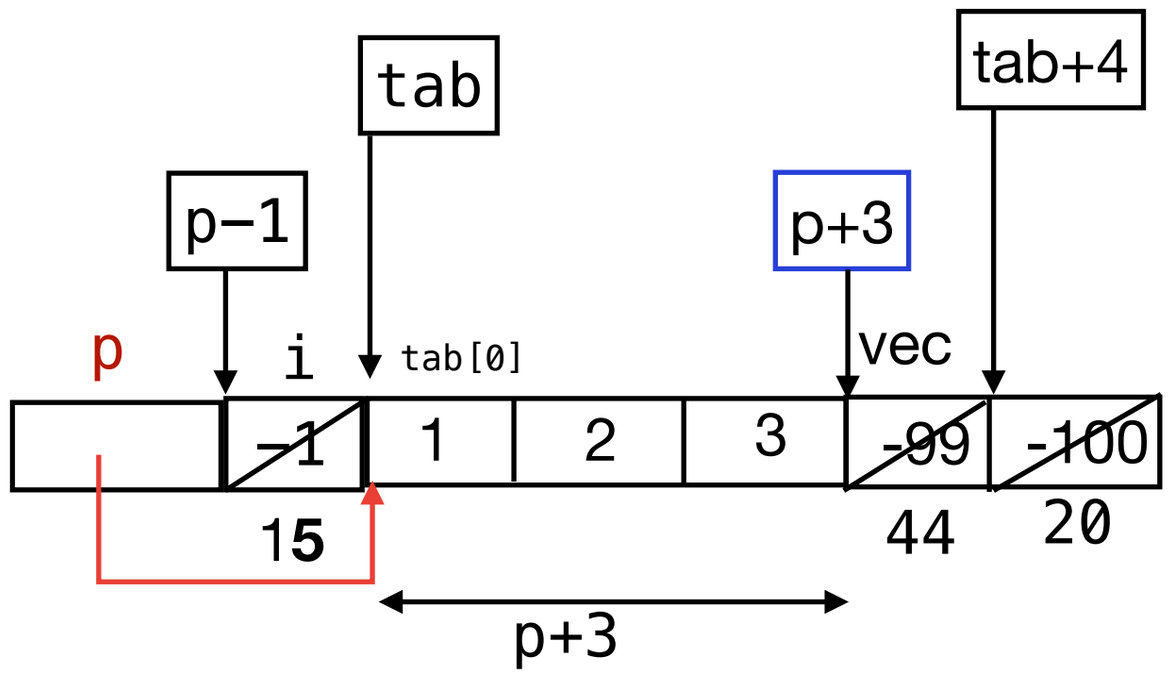

```
*(p+3) = 44 ;     /* equivalent à     p[3] = 44;        */
p[ i ] = 15;      /* equivalent à     *(p+i) = 15; */
tab[4] = 20;      /* equivalent à     *(tab+4) = 20 */
```

arithmétique de pointeurs

```
int vec[] = {-99, -100};

int tab[] = {1, 2, 3};

int i = -1;

int *p = &tab[0];

 printf("&p = %p \n&i = %p  \n&tab[0] = %p  \n&vec[0] = %p \n",

   &p, &i , &tab[0], &vec[0] );
```

Sur mon MacBook printf affiche :

```
&p = 0x7ffee28c5980
&i = 0x7ffee28c5988
&tab[0] = 0x7ffee28c598c
&vec[0] = 0x7ffee28c5998
```




```
*(p+3) = 44 ;      /* equivalent à          p[3] = 44;  */
p[ i ] = 15;       /* equivalent à   *(p+i) = 15; */
tab[4] = 20;       /* equivalent à   *(tab+4) = 20 */

printf(" i = %d \nvec[0] = %d\n\vec[1]=%d\n",
        i, vec[0], vec[1] );
```

Différence de deux pointeurs

arithmétique de pointeurs - différence de pointeurs

```
#include <stddef.h>

int tab[] = {1,2,3,4,5,6,7,8,9,10};  int i = 2; int j = 7;

int *pa = &tab[i];

int *pb = &tab[j];

ptrdiff_t  d = pa - pb;              --->  le même résultat que  i-j

printf("diff = %ld\n", (long) d);  --->  diff = -5, il y a 5 places
                                         de la taille sizeof(int)
                                         entre les adresses pa et pb
```

`ptrdiff_t` un type entier signé qui dépend de l'implémentation.

Le résultat de la différence de deux pointeurs `(du même type) est de type ptrdiff_t.` Le type `ptrdiff_t` défini dans `stddef.h`

La différence de deux pointeurs du même type alpha c'est la taille de la mémoire entre deux adresses **mesurée en nombre d'objets de type alpha** que nous pouvons stocker entre les deux adresses. La différence de deux pointeurs dépend de type de pointeurs.

Tableau comme paramètre de fonction

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

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

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

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

```
int main(void){
    int tab[]={4,6,8,9,11};
    double d = avg(5, tab);
}
```

Pendant l'appel de la fonction :

le paramètre `t` de la fonction `moy()` est une variable locale de la fonction initialisée avec l'adresse

`&tab[0]` (ou `tab`)

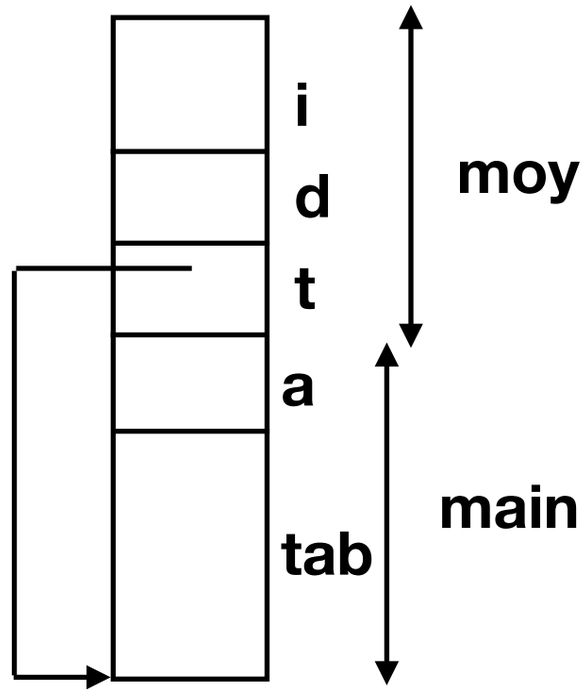
du premier élément du tableau tab.

En particulier **à l'intérieur de la fonction moy() :** `sizeof(t) == sizeof(int *)`

parce que le vrai type de t est int *.

réduction de tableau vers pointeur à l'appel de fonction

```c
double moy(size_t nb_elem, int t[]){     /* double moy(size_t nb_elem, int *t ){...} */

  double d = 0;

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

    d += t[i];      /* même chose que  d += *(t+i) ;   */
  }
  return d / nb_elem;
}

int main(void){

  int tab[] = {-6,7,66,-111,77,23,19,34,-89,45};

  double a = moy(sizeof tab / sizeof tab[0], tab);

  printf("%3.1f\n",a);

  a = moy(4, &tab[3]);  /* la moyenne des 4 éléments tab[3]...tab[6]  */

  printf("%3.1f\n",a);

  return 0;

}
```



Il est donc possible de calculer une fonction sur l'intervalle de tableau en passant l'adresse du premier élément de l'intervalle et le nombre d'éléments.

réduction de tableau vers pointeur à l'appel de fonction

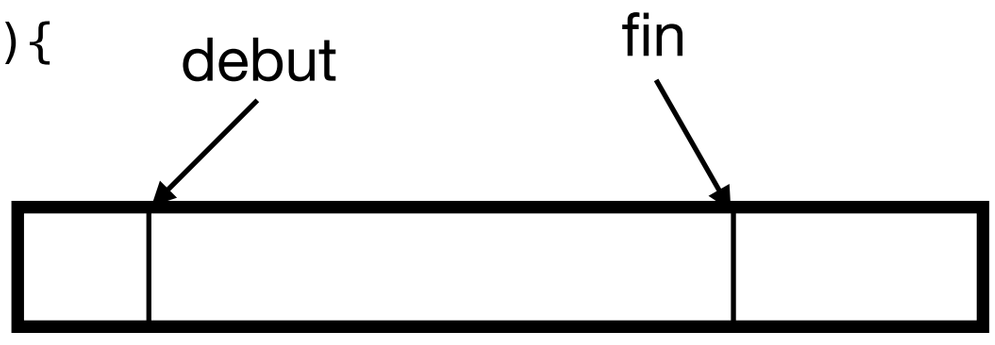
```
double moy( int nb_elem,  int t[]){
  int *fin = t + nb_elem; /* l'dresse juste après
                             le dernier élément */
  double d = 0;
  while( t < fin ){
    d += *t;
    t++;    /* t est une vraie variable et
             * peut changer valeur*/
  }
  return d / nb_elem;
}
```

réduction de tableau vers pointeur à l'appel de fonction

```
double moy( int nb_elem,  int t[]){
  int *fin = t + nb_elem; /* l'dresse juste après
                             le dernier élément */
  double d = 0;
  while( t < fin ){
    d += *(t++);     /* où même d += *t++;  */
  }
  return d / nb_elem;
}
```

exemple - recherche dichotomique dans un tableau trié

```
int *recherche(int *debut, int *fin, int a){
  int *x;
  while( fin - debut > 1 ){
    x = debut + (fin - debut )/2;
    if( *x == a)
      return x;
    else if( a < *x )
      fin = x ;
    else
      debut = x + 1;
  }
  return (debut < fin && *debut == a) ? debut : NULL;
}
```



la fonction retourne le pointeur vers élément a dans la valeur est a où NULL si la recherche échoue

debut - le pointeur vers le premier élément

fin - l'adresse juste après le dernier élément

exemple - recherche dichotomique dans un tableau trié

```
int *x, *debut, *fin;

x = debut + (fin - debut )/2;  /* OK */

x = (debut + fin)/2   /* incorrecte,
l'addition de deux pointeurs n'est pas
définie */
```

exemple - recherche dichotomique dans un tableau trié

```
int main(void){
  int t[] = {-12, -11, 6, 7, 23, 31, 33, 37, 43, 53, 57, 76, 79, 92, 99 };
  int taille = sizeof t/ sizeof t[0];
  int *r;
  r = recherche(&t[0], &t[taille], 33);  /*notez que t[taille] l'adresse
                                          du premier octets après le tableau t */
if( r != NULL )
   printf("element numero %ld\n", (long) ( r - &t[0] ));
 else
   printf("non trouve\n");

 r = recherche(&t[2], &t[12], 11);
  rechercher 11 parmi les entiers sur l'intervalle   t[2],...,t[11]
```