Langage C

Wieslaw Zielonka
zielonka@irif.fr

Annonces :

- à partir d'aujourd'hui le cours 16h00-18h00
- une interrogation pendant le cours le 17 février, la durée entre 30 et 45 minutes. Tous les sujets traités en TP jusqu'à cette semaine inclue (donc pas de pointeurs qui sont traités en cours d'aujourd'hui)
- inscrivez-vous en groupes TP sur moodle

#include <limits.h>

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

SHRT_MAX SHRT_MIN
INT_MAX -- valeur int minimale INT_MIN valeur int maximale
LONG_MAX LONG_MIN
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

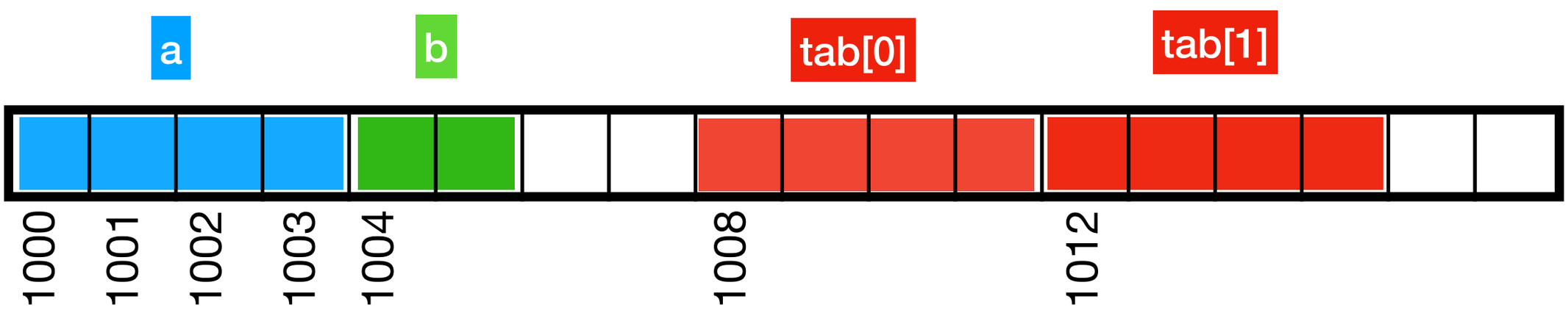
uint32_t entier sans signe de 32 bits $[0, 2^{32}-1]$

Pointeurs - arithmétique de pointeurs

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

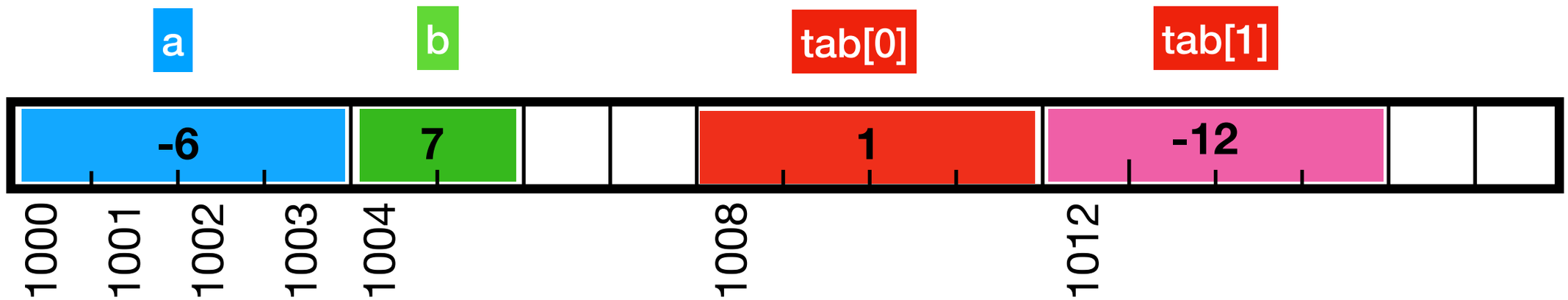
L'ordre de données dans la mémoire pas forcement le même que l'ordre de déclaration.

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 multiple de sizeof(int).

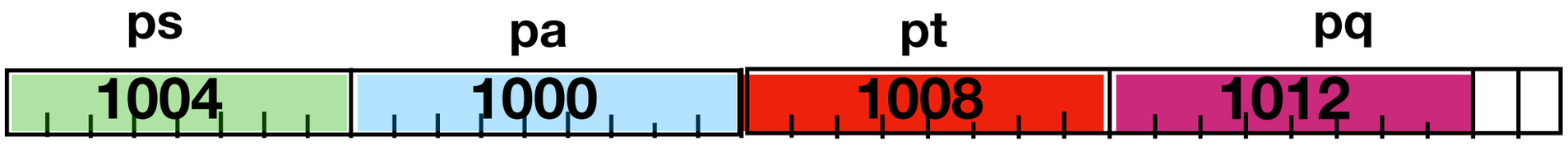
Pointeurs : les adresses
Opérateur &

```
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];
```

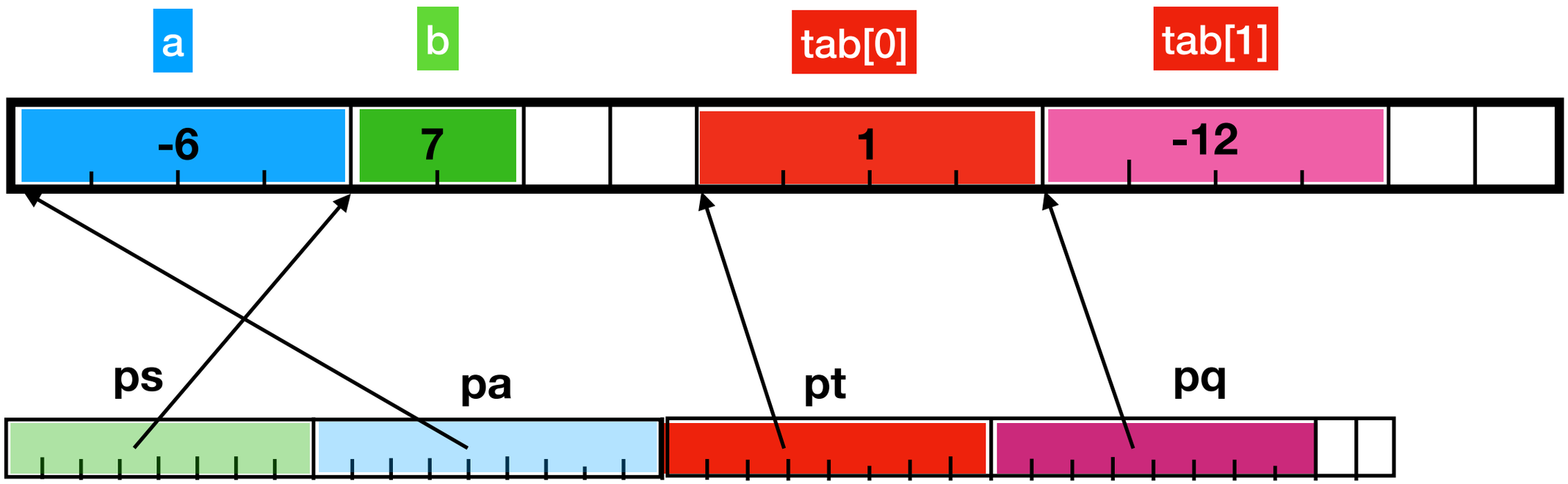


Sur mon portable sizeof(pointeur) == 8 donc 8 octets de la mémoire pour stocker une adresse (processeurs 64 bit = 8*8).

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
```



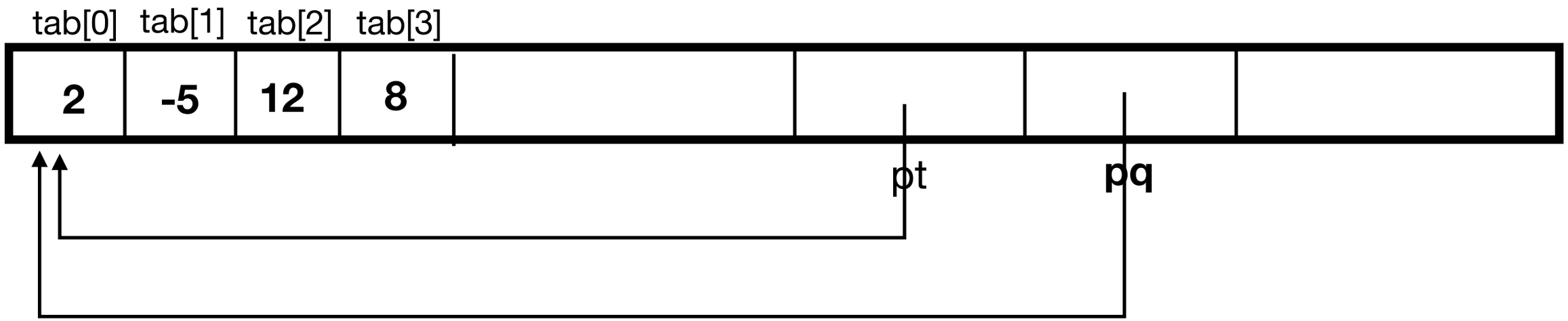
Attention : l'ordre réel de variables dans la mémoire peu être différent

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;  /* pas de & 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;
```

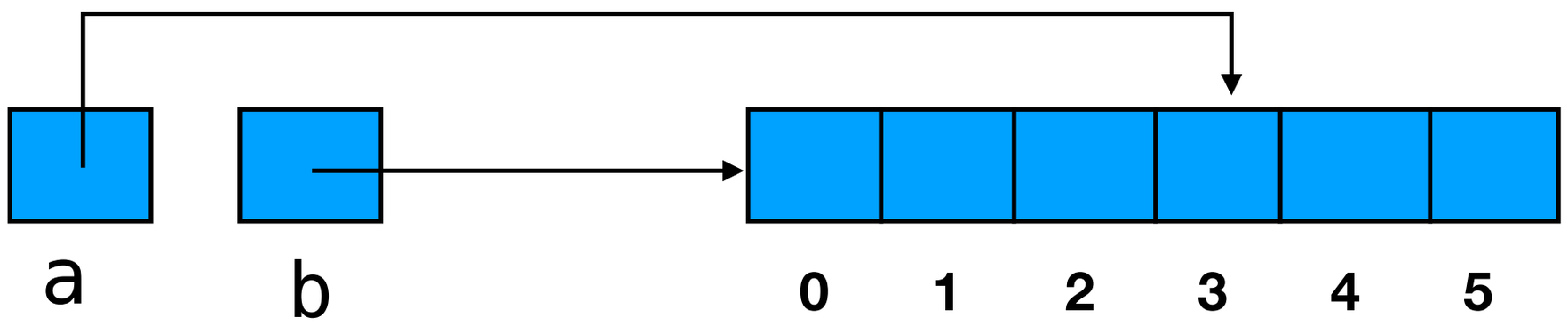
Deux variables pointeurs ***a, b*** de type ***int*** * déclarées d'un seul coup, différent de

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

c est un pointeur vers int,

d une variable int, pas un pointeur.

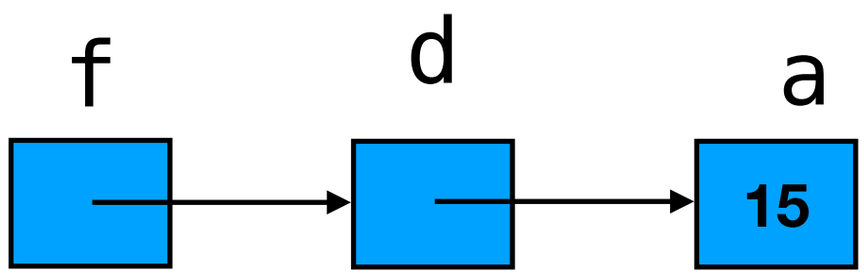
```
int tab[6];  a = &tab[3]; b = &tab[0];
```



pointeur de pointeur

```
int a=15;

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

int **f;   /* f un pointeur vers un "int *"  */
```

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



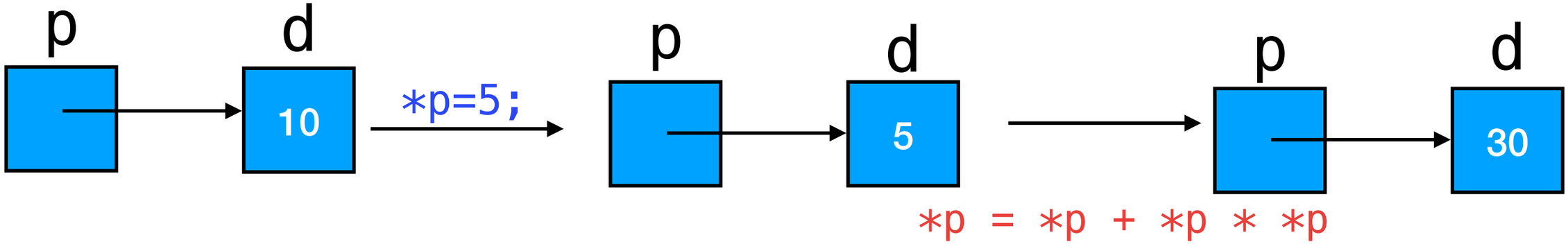
opérateur * appliqué au pointeur à gauche de l'affectation

```
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 */
```

opérateur * dans une expression

```
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
```

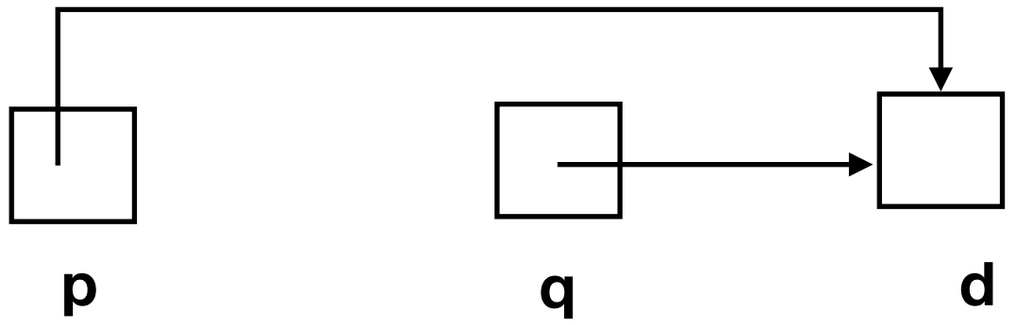


*p dans une expression c'est la valeur stockée à l'adresse p

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

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

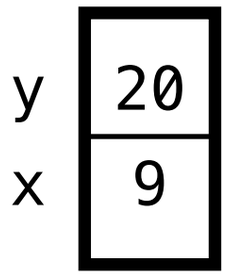
opérateur * appliqué à un pointeur

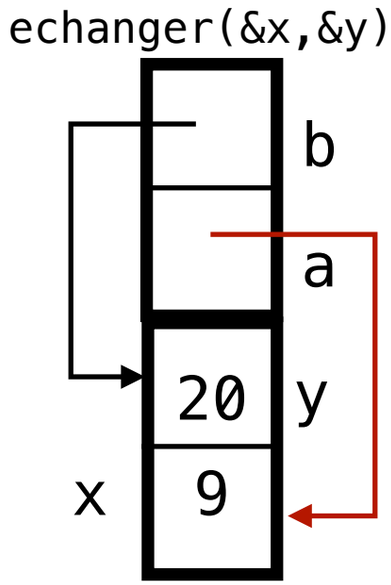



```
int d;

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     */

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 */
```

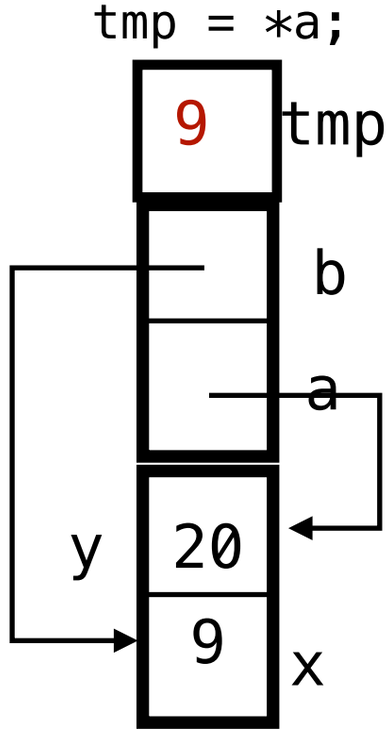
les pointeurs et les arguments de fonctions

```
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);

}
```

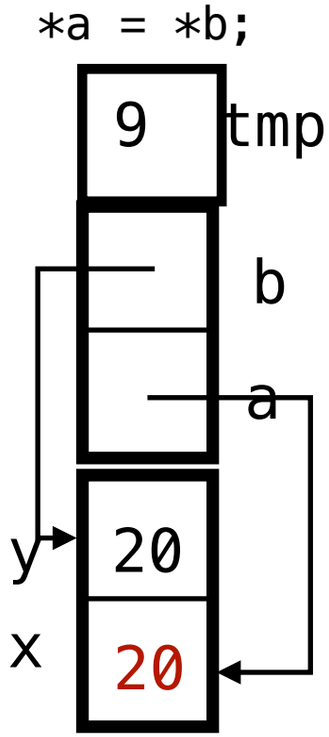




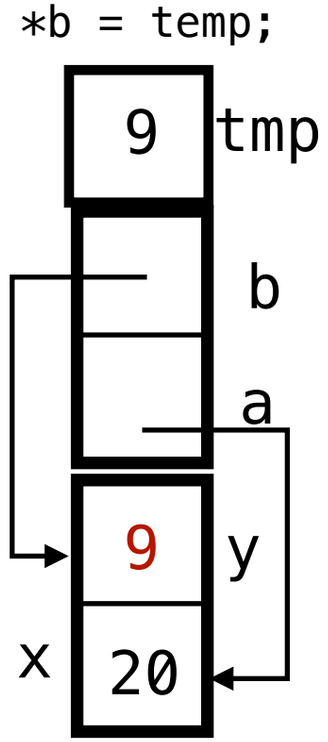
tmp = *a;



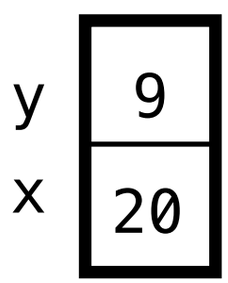
*a = *b;



*b = temp;



return



valeur 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

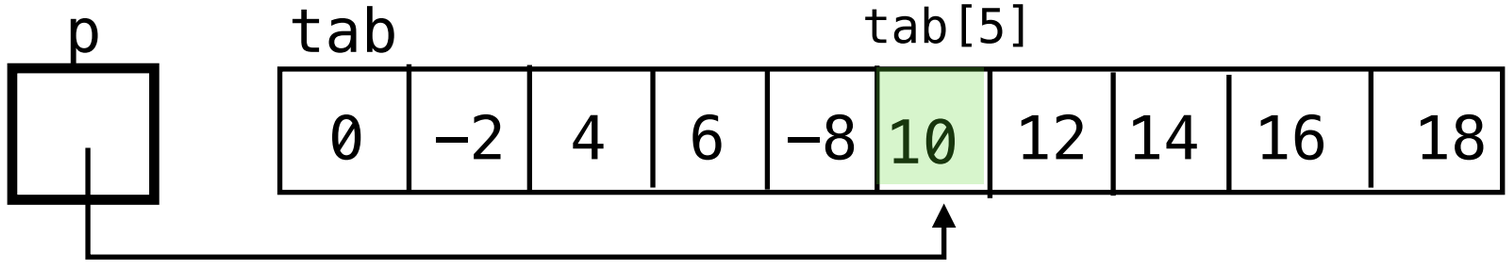
```
int tab[]={0,-2,4,6,-8,10,12,14,16,18};
int *p = &tab[5];
printf("%i \n", *p);
p = p + 3;
printf("%i \n", *p);
p = p - 5;
printf("%i \n", *p);
```

10

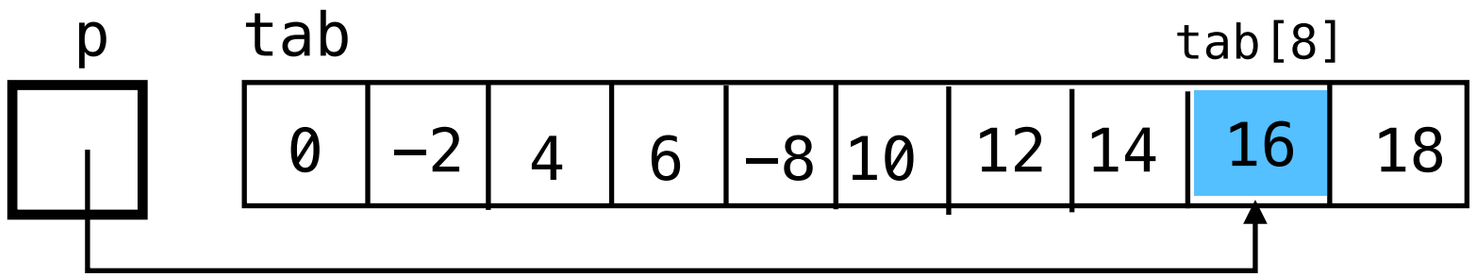
16

6

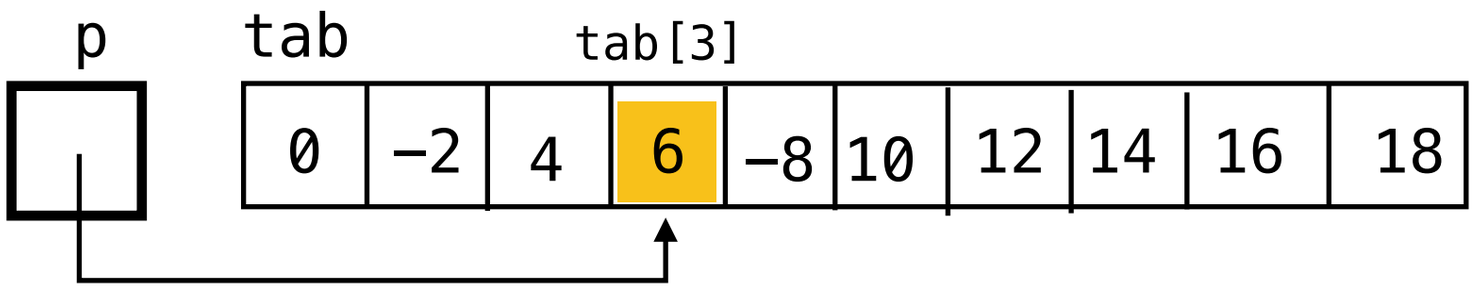
p = &tab[5];



p += 3;



p = p - 5;

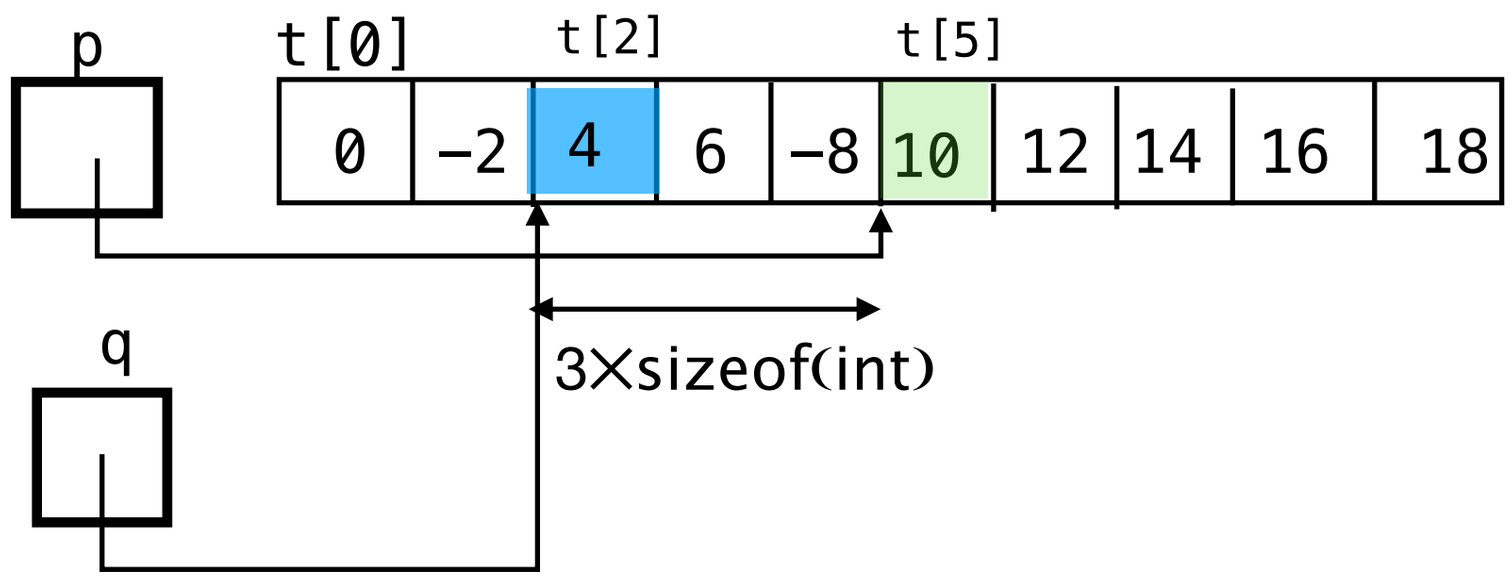


arithmétique de pointeurs

```
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 de type t :

t *p;

et n une expression de type int alors les valeurs des expressions

p + n et p - n

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

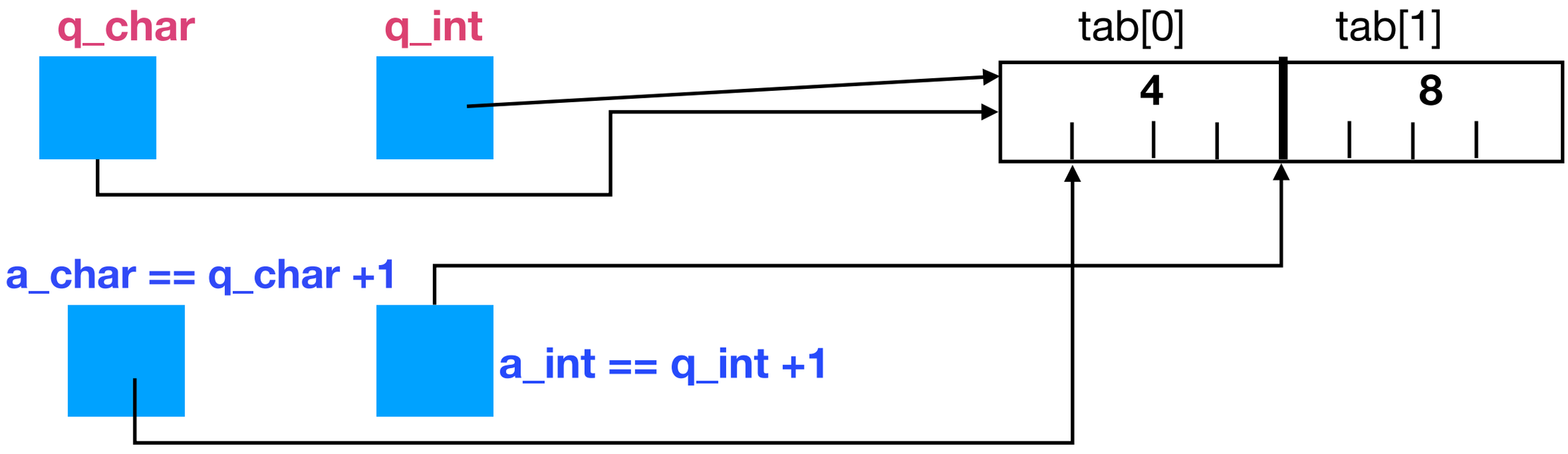
n * sizeof(t)

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


/* 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 == ???
```

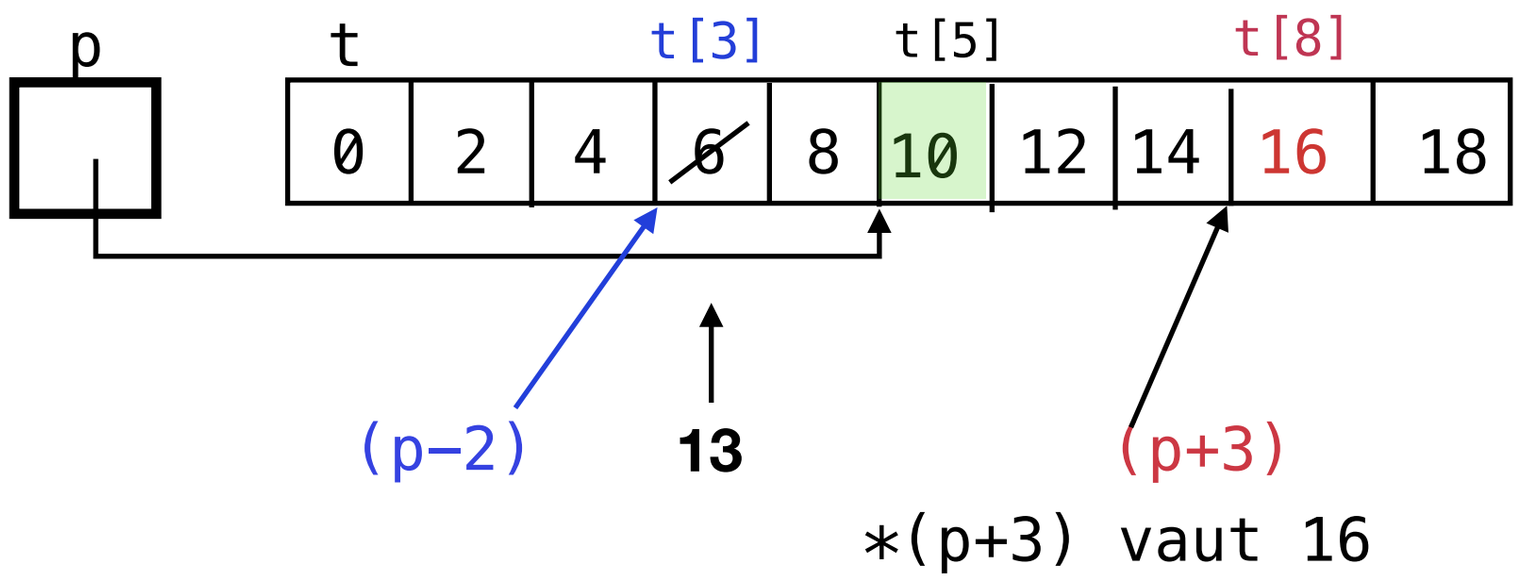


arithmétique de pointeurs

```
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 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".

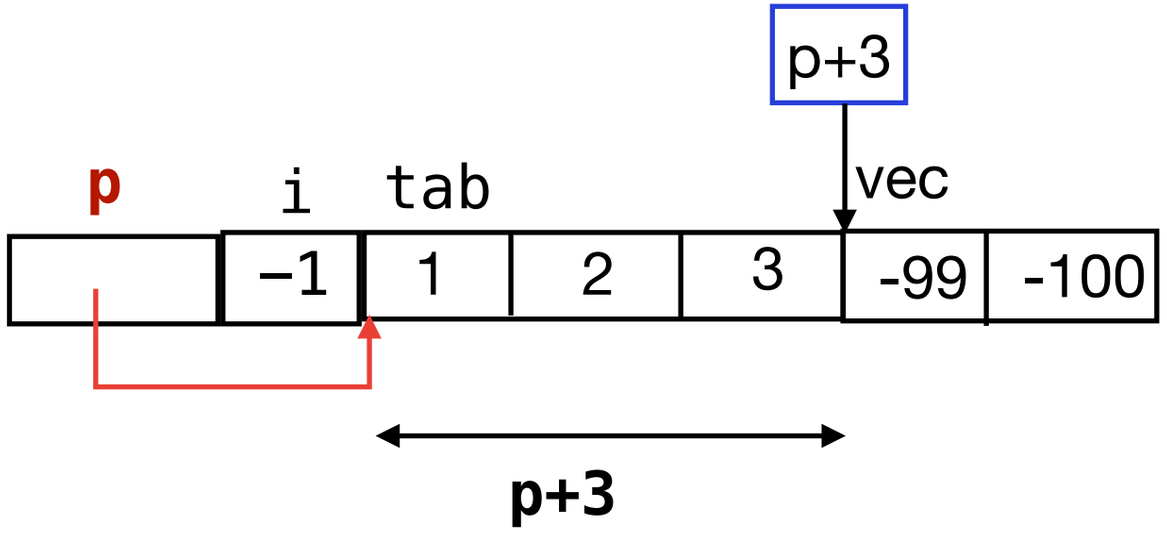
Exemple: le programme suivant, manifestement erroné, a été compilé et exécuté sur MacOS sans erreur ni warning (mais produit des résultats bizarres).

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

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

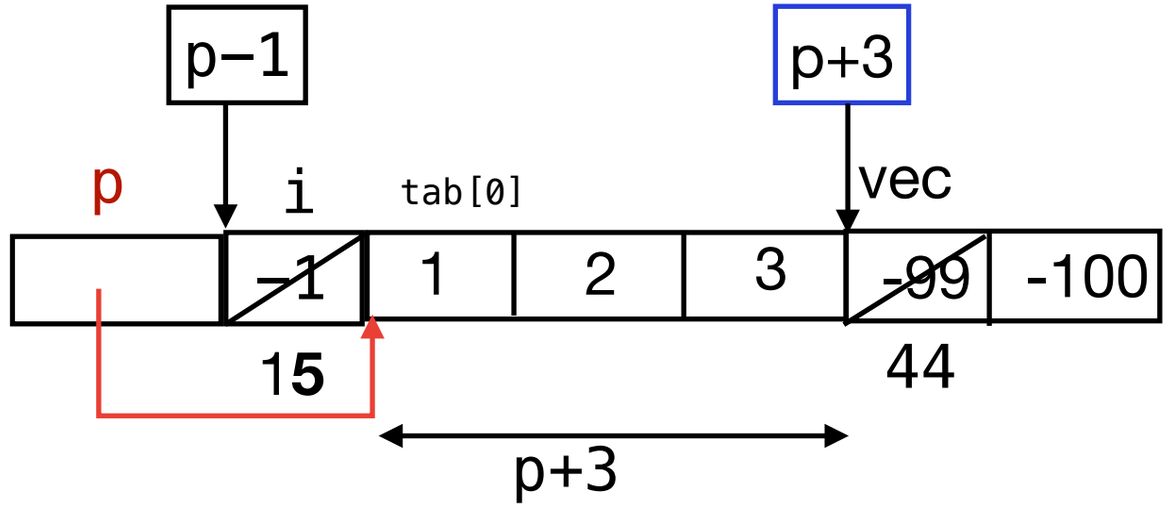



```
*(p+3) = 44 ;      /* equivalent à      p[3] = 44;        */
p[ i ] = 15;       /* equivalent à      *(p+i) = 15; */
```

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] );
```

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

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 = -7, il y a 7 places
                                         de la taille sizeof(int)
                                         entre les adresses pa et pb
```

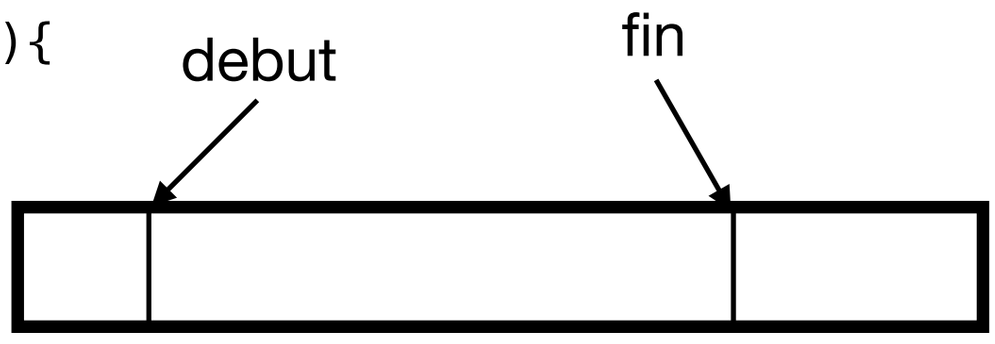
`ptrdiff_t` un type entier signé qui dépend de l'implémentation.

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.

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]
```

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 initialisé avec l'adresse

`&tab[0]`

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 lors de l'appel de fonction

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

  double d = 0;

  int i = 0;

  for(  ; 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]);  /* moyenne sur les éléments tab[3]...tab[6]  */

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

  return 0;

}
```

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 correcte?

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 dépilées donc somme() retourne l'adresse qui n'est plus valable.

Ce programme n'est pas correct.

pointeur générique : void *

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

int *p = &tab[1];

void *t=p;

char *c=t;
```

Nous pouvons faire une affectation entre un pointeur générique et un autre pointeur sans retypage "cast". C garantie que la valeur de pointeur est préservée.

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;`~~

scanf - lecture sur le terminal

scanf(), lecture sur le terminal

`scanf()` lit depuis les terminal et met les valeurs lues dans des variables, le premier paramètre de `scanf()` le format, **tous les paramètres suivant sont des pointeurs** donnant les adresses de variables où scant place les valeurs lues.

`scanf()` retourne les nombres d'éléments lus

```
int a,b;

int l = scanf("%i %i\n", &a, &b);

/* l sera 0 si la lecture échoue, 1 si la lecture de a
réussie mais pas de b, 2 si la lecture de a et b réussie
*/
```

scanf() quelques formats

format	le type de paramètre	remarques
%10s	char * lire jusqu'à 10 caractères, ou jusqu'à l'espace	%s la plus longue chaîne sans espaces (attention danger !)
%lf %lg %le	double *	
%f %g %e	float *	
%d %i	int *	%d accepte uniquement la notation décimal %i accepte décimal, octal, hexadécimal
%ld %li	long int *	
%c	un caractère	%5c 5 caractères
%x %o %u	unsigned int *	
%lo %lu %lx	unsigned long int *	

scanf() quelques formats

format	le type de paramètre	remarques
%hd %fi	short *	
%ho %hu %hx	unsigned short *	

scanf() exemple

```
int t[4];  unsigned int u;   double d[4];  char c[30];

int k = scanf("%d %i %x", &t[0], &t[1], &t[2]);

for(int i=0; i<k; i++)
    printf("t[%d]=%d\n", i , t[i]);
```

```
-89   010   0x100   999.99
t[0]=-89
t[1]=8
t[2]=256
```

(Enter)

entrée

sortie sur terminal

```
k = scanf("%le %lg %lf",  &d[0], &d[1], &d[2]);

for(int i=0; i<k; i++)
    printf("d[%d]=%4.2e\n", i , d[i]);
```

```
88e7 -77.777e3
d[0]=1.00e+03
d[1]=8.80e+08
d[2]=-7.78e+04
```

(Enter)

notez que 999.9 sera lu quand on envoie la deuxième ligne sur le termin et cette valeur sera arrondi à 1000

scanf()

Un caractère blanc c'est un caractère espace ou le caractère de nouvelle ligne.

- Un caractère blanc dans le format correspond à une suite quelconque de caractères blancs à l'entrée

- le caractère à l'entrée qui n'est pas "matché" par le format reste dans le tampon de l'entre, `scanf()` retourne sans lire la suite.

- `scanf()` retourne un entier : le nombre d'item qu'il a lu à l'entrée

scanf() exemple

```
int a, i;

for(i=0;i<5;i++){
  scanf("%d\n",&a);
  printf("valeur lue %d\n",a);
}
```

Sur le terminal j'entre 5 8 9 7 5 4. Notez que le dernier 4 c'est juste pour terminer l'entrée, cette valeur n'est pas affichée. Notez le décalage entre les entrées et les sorties.

```
5
8
valeur lue 5
9
valeur lue 8
7
valeur lue 9
5
valeur lue 7
4
valeur lue 5
```

scanf() exemple

```
int a, i;

for(i=0;i<5;i++){
  scanf("%d",&a);
  printf("valeur lue %d\n",a);
}
```

Sur le terminal:

```
6
valeur lue 6
8
valeur lue 8
9
valeur lue 9
7
valeur lue 7
6
valeur lue 6
```

Tout va bien! Les entrées sorties intercalées comme il faut.

Mais essayez de taper un autre caractère, par exemple x à la place d'un nombre:

```
3
valeur lue 3
7
valeur lue 7
x
valeur lue 7
valeur lue 7
valeur lue 7
```

Impossible de faire entrer quoi que se soit après x.

scanf() exemple

```
int a;
char c;
int i=5, k;
while(i){
  k = scanf("%d",&a);
  if(k == 0){
    scanf("%c",&c);
    continue;
  }
  i--;
    printf("valeur lue %d\n",a);
  }
}  6
   valeur lue 6
   x
   7
   valeur lue 7
   9
   valeur lue 9
   -66
   valeur lue -66
   qs
   fg
   99
   valeur lue 99
```

Lire cinq valeurs int.
Le programme marche correctement même quand l'utilisateur fait de fautes de frappe.