

Langage C

Wieslaw Zielonka
zielonka@irif.fr

allouer et libérer la mémoire

```
#include <stdlib.h>
```

```
void *malloc(size_t size)
```

malloc() alloue le mémoire de size octet et retourne le pointeur vers la mémoire allouée ou NULL en cas d'échec. La mémoire allouée n'est pas initialisée.

```
void *calloc(size_t count, size_t size)
```

calloc() alloue (count * size) octets de mémoire et retourne l'adresse de la mémoire allouée ou NULL. La mémoire est initialisée avec tous les octets à 0.

```
void *realloc(void *ptr, size_t size)
```

```
void free(void *ptr)
```

free() libère la mémoire allouée par une des fonctions ci-dessus.

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

`void *malloc(size_t size)`

Allouer un tableau de n nombres doubles:

```
double *tab = malloc( n * sizeof(double) );
if(tab == NULL){    /* toujours vérifier si malloc() réussit */
    perror("malloc");
    exit(1);
}
/* nous pouvons maintenant initialiser la mémoire pointée par tab */
for(int i = 0; i < n ; i++){
    tab[i] = ... ; /* même chose que *(tab+i) = ... ; */
}
```

`void perror(const char *s)` affiche un message d'erreur, à utiliser uniquement quand un appel fonction échoue, dans `<stdio.h>`

`void exit(int status)` termine l'exécution de programme avec le code status, dans `<stdlib.h>`

`exit(n) ;`

a le même effet que

`return n;`

dans la fonction `main()`.

perror()

```
#include <stdio.h>
```

```
void perror( const char *s )
```

perror() affiche un message d'erreur la chaîne de caractères s suivie par un message d'erreur.

perror() est à utiliser **uniquement** quand un appel fonction échoue.

exit()

```
#include <stdlib.h>
```

```
void exit( int status )
```

termine l'exécution de programme avec le code status

```
exit( n ) ;
```

a le même effet que

```
return n;
```

dans la fonction main().

Attention : comme les valeurs de exit() ou de retour de main() il faut utiliser uniquement les nombres entre 0 et 127.

```
exit( 0 ); /* le programme termine correctement */
```

```
exit( 1 ); /* valeur de exit > 0 signifie que le programme ne  
termine pas correctement */
```

```
exit( EXIT_SUCCESS );      exit( EXIT_FAILURE );
```

EXIT_SUCCESS et EXIT_FAILURE sont définis dans stdlib.h

EXIT_SUCCESS vaut 0 EXIT_FAILURE vaut 1

```
void *malloc(size_t size)
```

Une autre notation pour le calcul de taille de mémoire à allouer:

```
double *tab = malloc( sizeof( double [ n ] ) );
```

```
/*      sizeof( double [ n ] )  
*      équivalent à  
*      sizeof( n * sizeof( double ) )  
*/
```

utiliser assert pour vérifier une condition

```
#include <assert.h>
```

```
double *tab = malloc( sizeof( double [ n ] ) );
```

```
assert( tab != NULL );
```

```
assert( condition )
```

si la condition de assert() est fausse (dans le sens du C) alors

- assert affiche un message qui indique la condition qui échoue le test et le numéro de la ligne où se trouve assert correspondant dans le fichier source
- assert() fait ensuite appel à abort() pour terminer le programme

Dans le code professionnel assert ne doit pas être utilisé pour vérifier si l'appel de fonction est sans erreur. Mais nous nous autoriserons cet écart de bonnes pratiques de programmation C.



```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 double *fusion(double *debuta, double *fina,
5               double *debutb, double *finb,
6               unsigned int *len_resultat){
7
8     unsigned int nb = (fina-debuta) + (finb-debutb) ;
9     if( len_resultat != NULL )
10         *len_resultat = nb;
11
12     double *resultat = malloc( nb * sizeof(double) );
13
14     if( resultat == NULL )
15         return NULL;
16
17     double *current;
18     for ( current = resultat; debuta < fina && debutb < finb ; current++ ){
19         if( *debuta < *debutb ){
20             *current = *debuta;
21             debuta++;
22         }
23         else{
24             *current = *debutb ;
25             debutb++ ;
26         }
27     }
28
29     /* Terminer de recopier le tableau qui n'est pas terminé */
30     while( debuta < fina ){
31         *current = *debuta;
32         current++;
33         debuta++;
34     }
35     while( debutb < finb ){
36         *current = *debutb;
37         current++;
38         debutb++;
39     }
40
41     return resultat;
42 }
43
```

fusion de deux tableaux
triés de nombres doubles.

debuta, fina - pointeurs vers
le début et la fin du premier
tableau

debutb, finb - pointeurs vers
le début et la fin du deuxième
tableau

len_resultat : paramètre de
sortie, nombre d'éléments
dans le tableau résultat

la fonction retourne le pointeur
vers le premier élément de
tableau résultat




```
46 void print_tab_double(unsigned int taille, double tab[] ){
47     for (unsigned int i=0; i < taille; i++)
48         printf("%s%6.2f", i==0? "" : ", ", tab[i]);
49     printf("\n");
50 }
51
52
53 int main(void){
54     double ta[]={-445.6, -21.8, -6.99, -3.23, 1.12, 2.3 };
55     double tb[]={-256.9, -45.8, -22.99, -12.63, -6.64, 3.42, 9.3, 65 };
56     size_t lena = sizeof(ta)/ sizeof(ta[0]);
57     size_t lenb = sizeof(tb)/ sizeof(tb[0]);
58
59     /* le dernier paramètre de fusion l'adresse d'une variable */
60     unsigned int len;
61     double *t = fusion(ta, ta+lena, tb , tb + lenb, &len);
62
63     print_tab_double(len, t);
64     free(t);
65
66
67     /* fusionner les 4 derniers éléments de ta et tb */
68
69     t = fusion( ta + lena -4, ta + lena, tb + lenb - 4, tb + lenb, &len );
70
71     /* équivalent à
72     t = fusion( &ta[lena -4], &ta[lena], &tb[lenb - 4], &tb[lenb], &len );
73     */
74     print_tab_double(len, t);
75     free(t);
76
77     return EXIT_SUCCESS;
78 }
```

grave erreur à éviter à l'examen


Créer un tableau qui contient tous les éléments positifs de t().

```
int *positifs(int n, int t[]){
    int k = 0, j = 0;
    for( int i=0; i<n; i++){
        if( t[i]>0 )
            k++;
    }
    int res[ k+1 ];
    res[ k ] = -1;
    for( int i=0; i<n; i++){
        if( t[i]>0 )
            res[j++] = t[i];
    }
    return res;
}
```



Non, **cela n'a pas de sens**, le tableau res "disparaît" au moment de retour de la fonction positifs. L'adresse retournée n'est plus correcte.

```
int *positifs(int n, int t[]){
    int k = 0, j = 0;
    for( int i=0; i<n; i++){
        if( t[i]>0 )
            k++;
    }
    int *res = malloc( sizeof(int [k+1]));
    res[ k ] = -1;
    for( int i=0; i<n; i++){
        if( t[i]>0 )
            res[j++] = t[i];
    }
    return res;
}
```



OK, mémoire allouée par malloc() reste allouée jusqu'à l'appel à free().

```
void *realloc(void *ptr, size_t size)
```

`realloc()` "modifie" la taille de la zone mémoire dont l'adresse est `ptr`

ptr : l'adresse valide d'une zone de mémoire retourné auparavant par `malloc()` `calloc()` ou `realloc()` (zone mémoire dans le tas, pas sur la pile)

size : la taille demandée en octets . Cette taille peut être plus grande ou plus petite que taille de la zone à l'adresse `ptr`. Les données qui se trouvent à l'adresse `ptr` sont recopiées dans la mémoire nouvellement allouée.

Si `realloc()` réussit à allouer la mémoire :

1. l'adresse **ptr** devient invalide (en particulier **ne faites plus `free()` sur `ptr`** et n'utilisez plus la mémoire à l'adresse `ptr`, `realloc()` libère lui-même la mémoire à l'adresse `ptr`),
2. `realloc()` retourne l'adresse du premier octet de la nouvelle zone mémoire.
3. `realloc()` recopie les données de l'adresse `ptr` vers la nouvelle adresse

Si `realloc()` échoue il retourne `NULL` et l'adresse `ptr` reste valide.

`realloc(NULL, n)` est équivalent à `malloc(n)`

comment utiliser realloc()

```
int *tab = malloc( n * sizeof(int) );

/* plus tard : si la taille de tab trop petite doubler la taille
de tableau */

int *p = realloc(tab, 2 * n * sizeof(int) );

if( p == NULL ){    /* toujours vérifier si realloc() a échoué */

    /* traitement d'erreur de realloc, tab reste valable */

} else{    /* realloc() réussit, tab n'est plus valide */

    tab = p ;

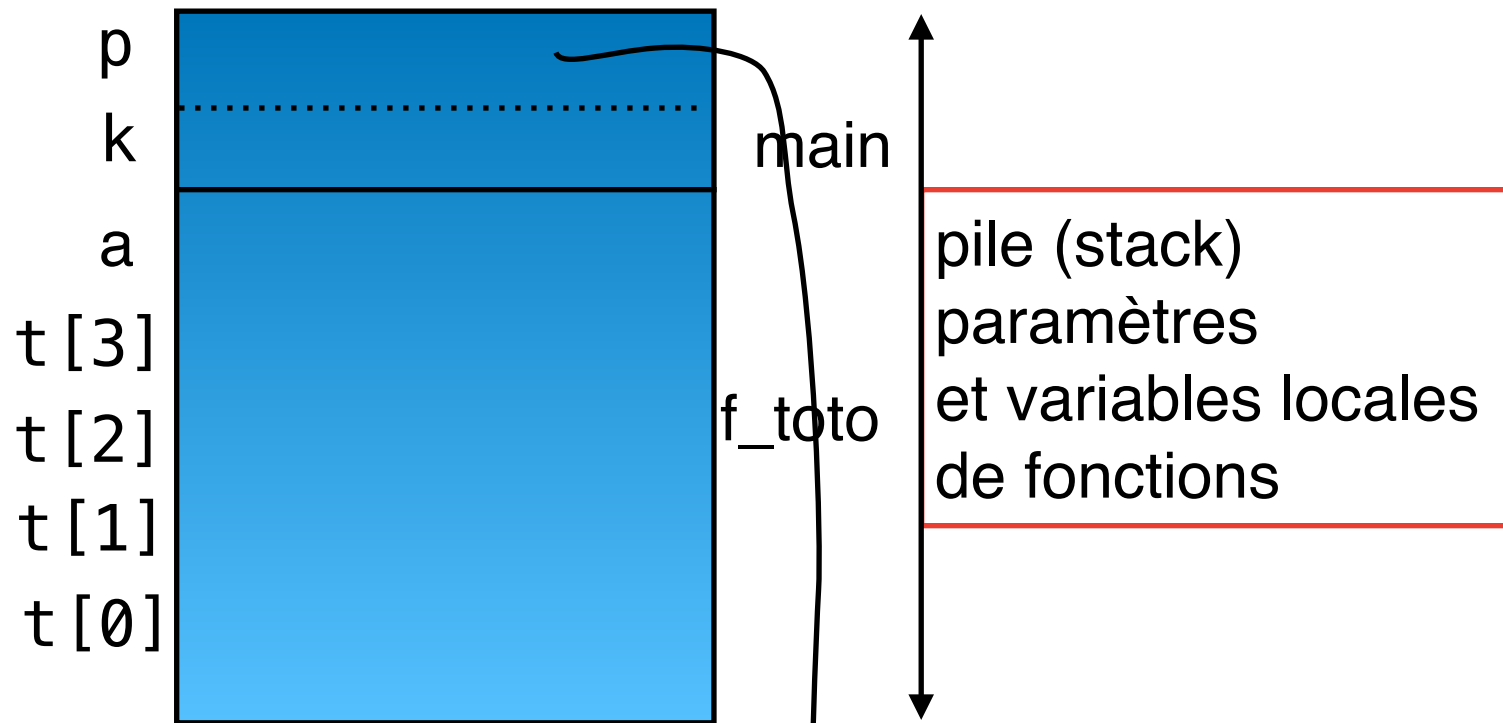
    n *= 2; /* deux fois plus d'éléments dans tab */

}
```

- soit tab n'a pas changé (si realloc a échoué et retourne NULL),
- soit p pointe vers la mémoire deux fois plus grande, donc la première moitié est initialisée avec les valeurs recopiées depuis "ancien" tab

mémoire d'un processus

les adresses mémoire les plus grandes



```
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);  
    .....  
}
```

variables globales et static

text segment

text segment contient le code binaire de programme

copier une zone de mémoire

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

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

memmove() copie n octet de l'adresse src vers l'adresse dst. La fonction retourne dst. Les deux zones de mémoire peuvent chevaucher.

```
int tab[] = {4, 7, 9, -12, 7, 8, 22};
```

```
int ptr = malloc( 5 * sizeof(int) );
```

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

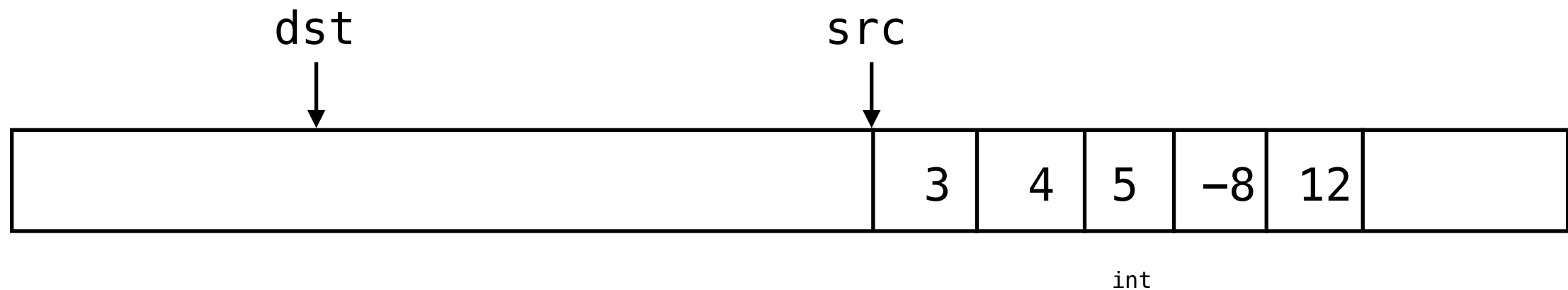
```
memmove( ptr , &tab[2], 5 * sizeof( int ) );
```

copier 5 derniers int de tab à l'adresse ptr

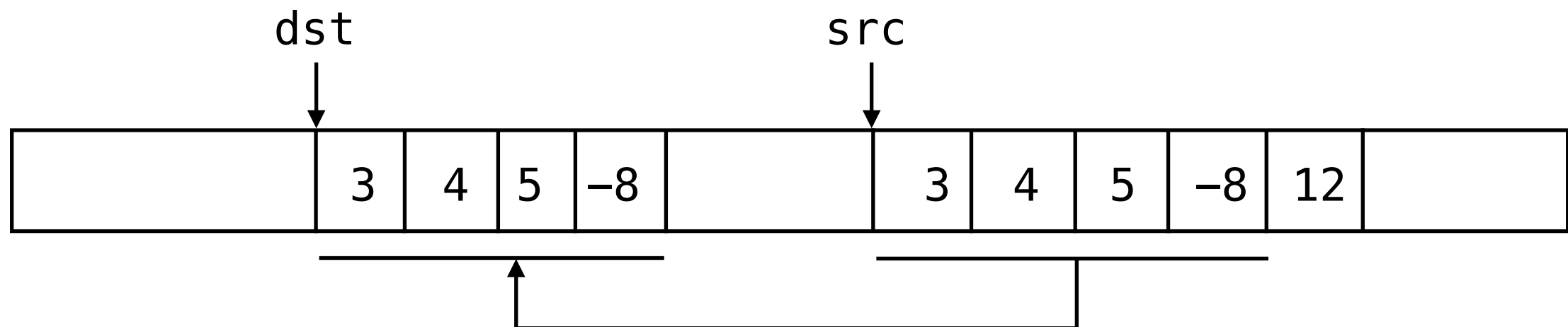
memmove

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

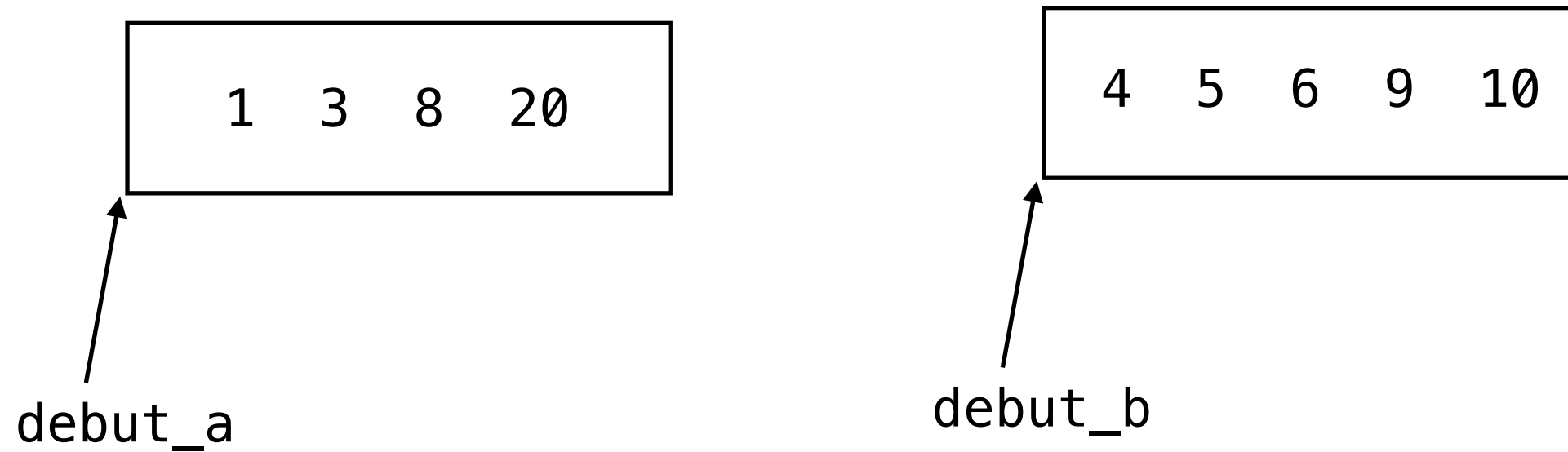
```
void *memmove(void *dst, const void *src, size_t n)
```



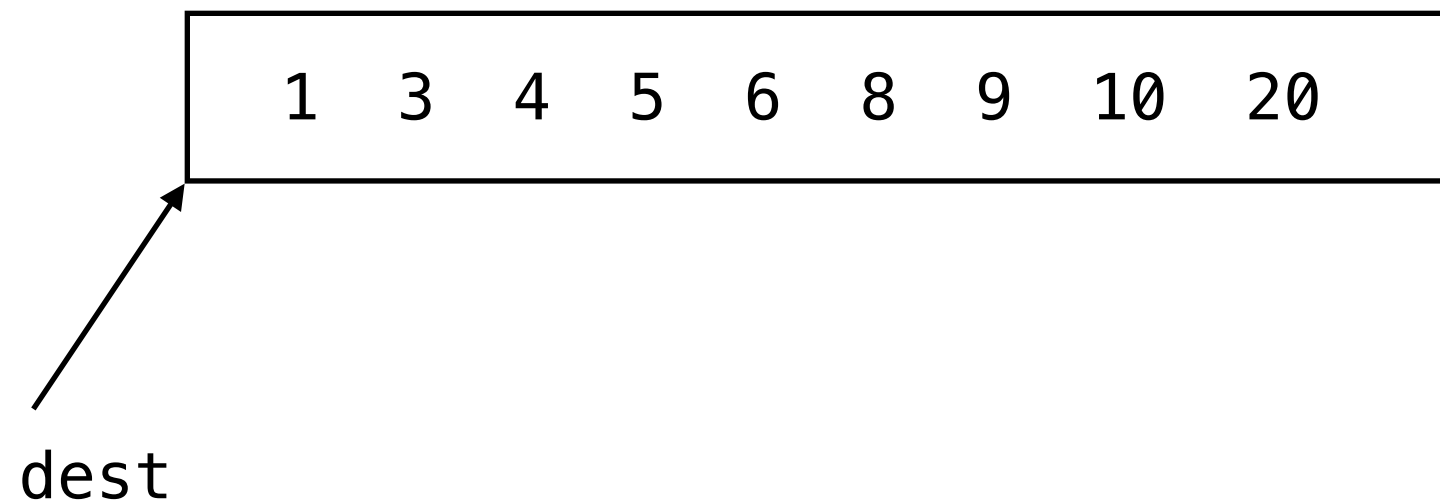
```
memmove( dst, src, 4*sizeof(int) )
```



exemple : fusion de deux tableaux triés de int



```
fusion(int *debut_a, unsigned int len_a,  
       int *debut_b, unsigned int len_b, int *dest)
```



exemple : fusion de deux tableaux triés

On suppose que la mémoire dest pour le résultat est allouée avant l'appel de la fonction fusion(). a, b deux tableaux triés, len_a, len_b le nombre d'éléments dans chaque tableau. On suppose que l'adresse dest pointe vers une zone de mémoire assez grande pour stocker le résultat – le tableau obtenu pas la fusion de a et b */

```
void fusion(unsigned int len_a, int *a,  
            unsigned int len_b, int *b, int *dest){
```

```
    unsigned int i, j;
```

```
    for( i=0, j=0 ; i < len_a && j < len_b; dest++ ){
```

```
        if( a[i] <= b[j] ){  
            *dest = a[i] ;  
            i++ ;
```

```
        }
```

```
        else{
```

```
            *dest = b[j] ;  
            j++ ;
```

```
        }
```

```
    }
```

```
    /* copier les éléments qui restent à la fin de tableau, soit a soit b */  
    /* si le dernier paramètre de memmove() est 0 memmove ne fait rien */
```

```
    memmove(dest, a + i, sizeof(int) * (len_a - i));
```

```
    memmove(dest, b + j, sizeof(int) * (len_b - j));
```

```
}
```

exemple : fusion de deux tableaux triés

```
/* main() pour tester fusion() */

int main(){
    int ta[]={-3, -8, 99, 120, 500};

    int tb[]={-100, -2, 40, 155};

    /* allocation de mémoire pour le résultat */

    int *tab = malloc( sizeof(ta) + sizeof(tb) );
    assert( tab != NULL );

    size_t lena = sizeof ta / sizeof ta[0] ;

    size_t len_b = sizeof tb / sizeof tb[0] ;

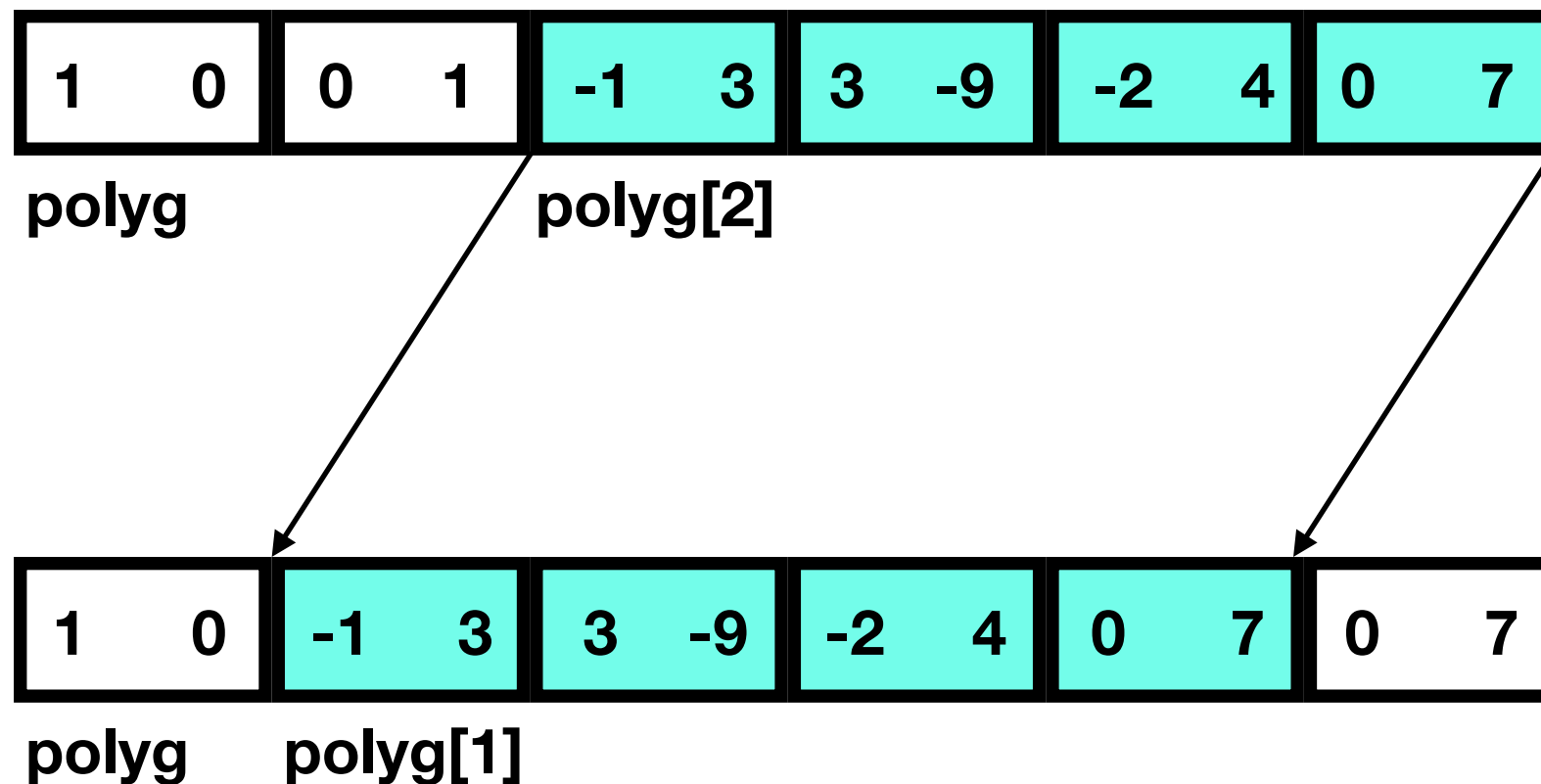
    fusion(ta, len_a, tb, len_b, tab);
    .....
}
```

memmove()

memmove() permet de copier une zone mémoire à l'intérieur de même tableau.

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

```
point polyg[] = { {.x=1}, {.y=1}, {.x=-1, .y=3},  
                 {.x=3, .y=-9}, {.y=4, .x=-2}, {.y=7} };
```



```
memmove( &polyg[1], &polyg[2], sizeof(point) * 4 );
```

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

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

memcpy() copie n octet de l'adresse src vers l'adresse dst. La fonction retourne dst. Les deux zones de mémoire **ne peuvent pas chevaucher**.

Dans l'exemple précédent on ne peut pas utiliser memcpy().

Consigne : préférer memmove() surtout en cas de doutes.

remplir une zone de mémoire

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

```
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

En pratique memset sert presque exclusivement pour mettre la valeur 0 dans une zone mémoire :

```
#define N 1024
```

```
int tab[ N ];
```

```
memset( tab, 0, N * sizeof(int) ); /* à la place d'une boucle qui met 0 dans tous
```

```
    * les élément de tab */
```