

Langage C

Wieslaw Zielonka
zielonka@irif.fr

```
#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

`malloc()`, `calloc()`, `realloc()` retournent l'adresse de la mémoire allouée ou `NULL` si l'appel échoue

```
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 n nombres doubles:

```
double *tab = malloc( n * sizeof(double) );
if(tab == NULL){    /* toujours vérifier si malloc() réussit */
    perror("malloc");
    exit(1);
}

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>`

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

Une autre notation :

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

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

`assert(condition)` si la condition est fausse (dans le sens du C) alors l'affichage d'un message qui donne le nom de fichier source et la ligne dans le code, et le programme termine

mais à quoi sert malloc() ?

```
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;
}
```

```
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;
}
```

```
int main(){
    int tab[] = { -5, 6, -9, 11, -3};
    int n = sizeof(tab)/sizeof(tab[0]);
    int *r = positifs( n, tab );
    for( int i = 0; r[i] > 0; i++ )
        printf("d\n", r[i]);
}
```

mais à quoi sert malloc() ?

```
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;
}
```

INCORRECT, res sur la pile

Le tableau res sur la pile, l'utilisation après return de la fonction n'est pas valide.

```
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, la mémoire allouée par malloc() n'est pas sur la pile mais dans le tas (heap) et reste utilisable jusqu'à la libération par l'appel à free()

```
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 de la zone mémoire allouée ou `NULL` en cas d'échec.

```
long *tab = calloc( n, sizeof(long));
```

```
/* tab – tableau de n éléments long  
 * 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.

fuites de mémoire

```
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;
}
```

```
int main(){
    int tab[ N ] ;
    int *r;

    while(.... ){
        /* remplir tab */

        r = positifs( n, tab );
        /* travailler sur r */

    }

}
```

Pas de ramasse-miette en C, la mémoire allouée par malloc() reste allouée jusqu'à l'appel à free. Si on perd l'adresse de la mémoire allouée impossible de la libérer. Deux bonnes pratiques :

1. faire malloc() et free() dans la même fonction, ou
2. pour chaque fonction qui alloue la mémoire écrire une fonction correspondante qui libère la mémoire.

fuites de mémoire


```
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;
}
```

```
int main(){
    int tab[ N ] ;
    int *r;

    while(.... ){
        /* remplir tab */

        r = positifs( n, tab );
        /* travailler sur r */

        detruire_positifs( r );
    }
}
```



```
void detruire_positifs( int *t ){
    free( t );
}
```

```
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 la 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.

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

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

exemple d'utilisation de realloc()

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

....

/* doubler la taille du tableau */

n *= 2;

int *p = realloc(t, n * sizeof(int) );

if(p == NULL){

    /* realloc() a échoué et t reste valide */

    .....

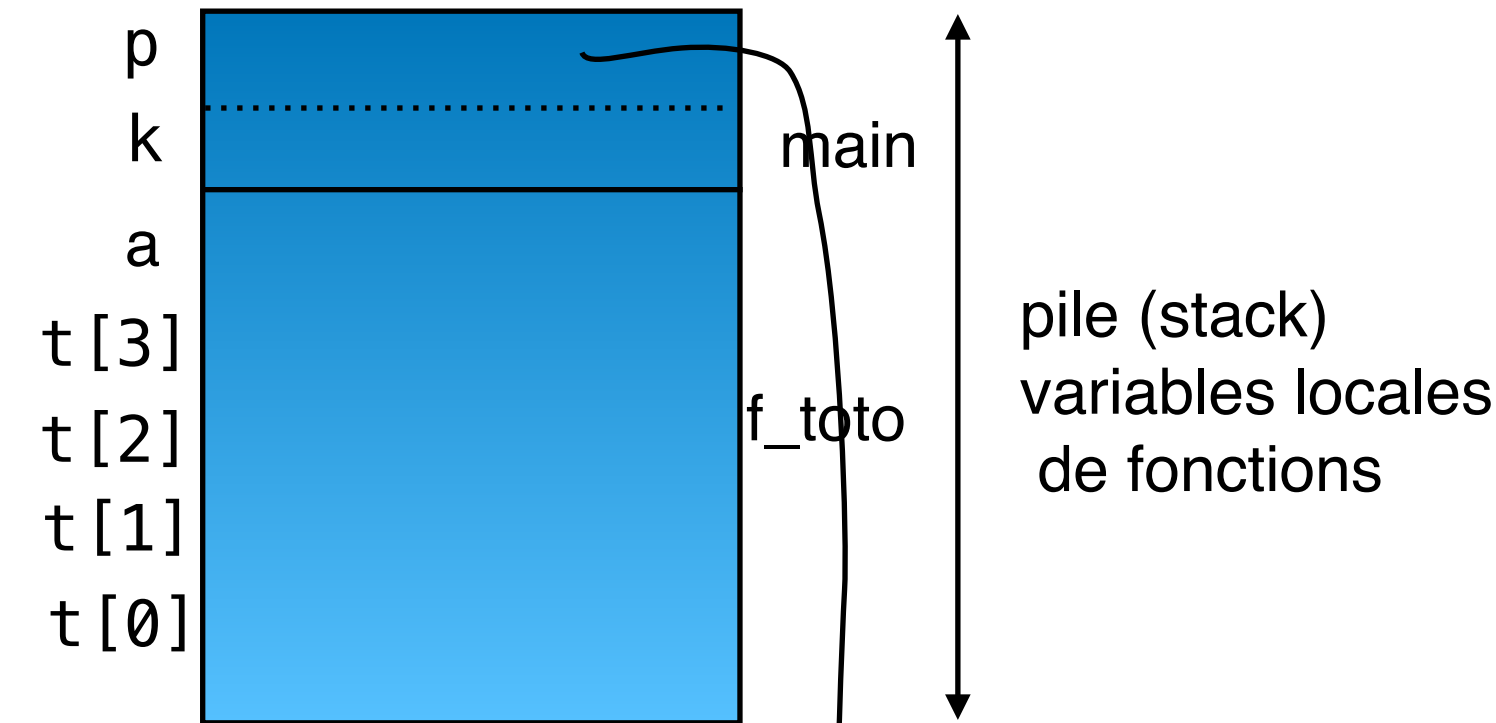
    exit( 1 ) ; /* exit() si on n'a rien à faire en cas de problèmes*/

}

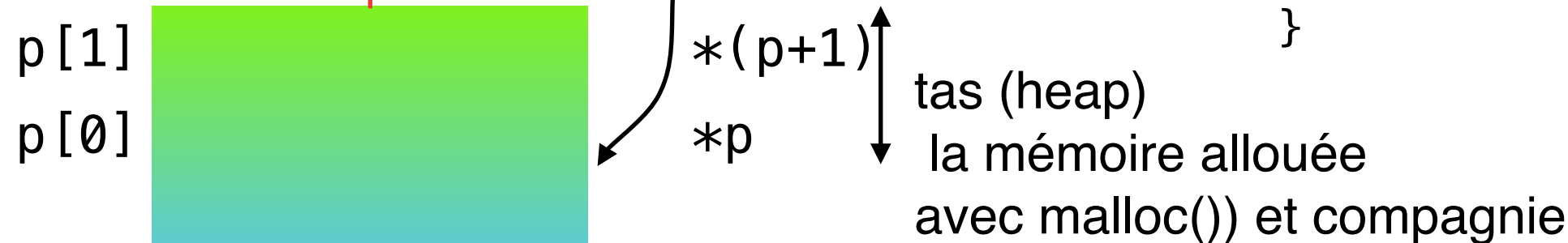
t = p ; /* si p != NULL alors t n'est plus valide on peut réaffecter */
```

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

le segment qui contient le code binaire de processus

adresses mémoire les plus petites

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

```
if( t == NULL )
```

```
    exit(1);
```

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

copie 5 derniers éléments de tab dans t

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 cela sert presque exclusivement pour mettre 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 */
```