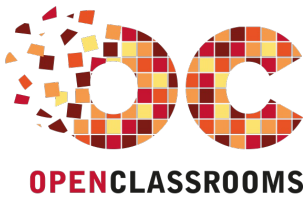
Allocation dynamique en C - complément

Par Adrien31100



www.openclassrooms.com

Dernière mise à jour le 12/05/2011

Sommaire

Allocation dynamique en C - complément

Par  Adrien31100

Mise à jour : 12/05/2011
Difficulté : Facile Durée d'étude : 45 minutes

Salut à vous amis Zéros, et bienvenue sur mon premier tutoriel !

Je vais vous parler ici des deux autres fonctions d'allocation dynamique de la mémoire :
calloc() et **realloc()**.

Ces fonctions sont très utiles, particulièrement realloc() comme vous pourrez le voir plus loin.
Sommaire du tutoriel :

La fonction calloc()

calloc : prototype, fonctionnement

La fonction calloc() a le même rôle que malloc(). Elle permet d'allouer de la mémoire. La différence entre les fonctions calloc() et malloc(), c'est que **calloc() initialise à 0 tous les éléments de la zone mémoire**.
calloc = Clear (memory) ALLOCation
Il faut faire `#include <stdlib.h>` pour pouvoir l'utiliser.

Voici son prototype :

Code : C

```
void* calloc(size_t num_elements, size_t size);
```

Le premier argument est le nombre d'éléments qu'on souhaite pouvoir stocker en mémoire et le deuxième est la taille de ces éléments que l'on obtient avec l'opérateur `sizeof()`.

En cas d'échec, elle renvoie un pointeur nul. Il est nécessaire de tester cette valeur pour ne pas faire d'opérations illégales.

Dans le code suivant, nous allons allouer 15 cases mémoires pouvant contenir des caractères et initialiser chacune de ces cases à zéro à l'aide de la fonction malloc():

Code : C

```
char *maChaine = malloc(15 * sizeof(char)); /* Ici, on ne connaît
pas la valeur des cases mémoires. La valeur de chacune des cases
mémoires est totalement aléatoire. */
int i;
```

```c
if (maChaine == NULL) { /* On vérifie que le système n'a pas
renvoyé un pointeur nul. */
    puts("ERREUR : probleme de memoire.");
    exit(EXIT_FAILURE);
}

for(i = 0; i < 15; i++)
    maChaine[i] = 0; /* On met chacune des cases à 0. */
/* suite du programme */
```

La même opération en utilisant calloc() donne le code suivant :

Code : C

```c
char *maChaine = calloc(15, sizeof(char)); /* Ici, on sait que les
15 cases mémoires contiennent 0. */

if (maChaine == NULL) {
    puts("ERREUR : probleme de memoire !");
    exit(EXIT_FAILURE);
}

/* suite du programme */
```

C'est un peu plus concis non ?

Un exemple

Pour bien voir la différence des fonction malloc() et calloc(), nous allons allouer de la mémoire avec chacune des deux fonctions et on va afficher leur contenu :

Code : C

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

int main()
{
    float *ptr1 = NULL, *ptr2 = NULL;
    int i, n = 5, termination = 1;

ptr1 = calloc(n, sizeof(float)); /* On alloue de la mémoire avec
calloc() pour ptr1. */
    if (ptr1 == NULL)
        printf("Echec de calloc().\n");
    else {
ptr2 = malloc(n * sizeof(float));/* On alloue de la mémoire avec
malloc() pour ptr2. */
        if (ptr2 == NULL)
            printf("Echec de malloc().\n");
        else {
            printf("Allocation de memoire avec\n");
            printf("calloc() pour ptr1\n");
            printf("malloc() pour ptr2.\n\n");
            for (i=0; i<n; i++)
                printf("ptr1[%d]=%f, ptr2[%d]=%f\n", i, ptr1[i], i,
ptr2[i]);
            /* On affiche la valeur de chacune des cases des
pointeurs. */
            printf("\n");
            free(ptr2);
            termination = 0;
        }
        free(ptr1);
```

```
        }

        return termination;
    }
```

Ce code peut générer par exemple :

Code : Console

```
Allocation de memoire
calloc() pour ptr1
malloc() pour ptr2

ptr1[0]= 0.000000, ptr2[0]= 0.000000
ptr1[1]= 0.000000, ptr2[1]= 0.000000
ptr1[2]= 0.000000, ptr2[2]= 242323749339136.000000
ptr1[3]= 0.000000, ptr2[3]= 706960599279049860000000000000.000000
ptr1[4]= 0.000000, ptr2[4]= 42061276.000000

Appuyez sur une touche pour continuer...
```

On peut voir sur cet exemple que toutes les cases de ptr1 sont effectivement à 0 alors que ptr2 pointe vers des valeurs quelconques qui peuvent être 0 comme pour ptr2[0] et ptr2[1].

La fonction realloc()

realloc : prototype, fonctionnement

realloc() s'utilise après qu'on ait utilisé la malloc() ou calloc(), mais on peut aussi la rappeler plusieurs fois de suite (dans une boucle **for** ou **while** par exemple).
Elle sert à ré-attribuer de la mémoire à un pointeur mais pour une taille mémoire différente.
Il faut faire `#include <stdlib.h>` pour pouvoir s'en servir.
Voici son prototype :

Code : C

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

Le premier argument est le pointeur sur lequel on désire effectuer l'opération, le deuxième argument est la taille de l'espace mémoire qu'on veut allouer.

realloc() modifie la taille de la zone mémoire précédemment attribuée à un pointeur soit par malloc(), soit par calloc() (la fonction utilisée avant l'appel de realloc()), soit par realloc() elle-même, et renvoie l'adresse de la nouvelle zone mémoire allouée.
Si la zone mémoire précédemment allouée peut être augmentée sans empiéter sur une zone mémoire utilisée pour autre chose, alors l'adresse mémoire renvoyée n'est pas modifiée (c'est la même que l'ancienne) mais le nombre de cases mémoires disponibles est modifié (c'était le but recherché).
En revanche, si en augmentant la zone mémoire initiale on déborde sur une zone mémoire déjà occupée, le système d'exploitation cherchera une autre adresse pour laquelle le nombre de cases mémoire nécessaires (le paramètre size) est disponible.

Comme ses demi-sœurs, elle renvoie un pointeur nul en cas d'échec. Il faut là aussi tester cette valeur si on ne veut pas risquer de faire planter notre programme.

un exemple

Code : C

```
char *maChaine = calloc(15, sizeof(char));
if (maChaine == NULL)
```

```
    pb_memoire(); /* Cette fonction affiche un message d'erreur et
termine le programme.
quelques instructions */

maChaine = realloc(maChaine, 20 * sizeof(char))
if (maChaine == NULL)
    pb_memoire();
/* quelques instructions */

free(maChaine);
```

La ligne 6 de l'exemple demande à attribuer 20 cases mémoire pouvant contenir un char à maChaine et retourne l'adresse pour laquelle ces 20 cases mémoires sont disponibles à maChaine.

La fonction pb_memoire() ressemblerait à quelque chose comme :

Code : C

```
void pb_memoire(void)
{
    printf("ERREUR : probleme de memoire !\n");
    exit(EXIT_FAILURE);
}
```

De plus, en cas de changement d'adresse du pointeur, les éléments stockés précédemment en mémoire sont déplacés vers la nouvelle adresse. Donc, dans l'exemple précédent, les cases 0 à 14 contiennent les caractères qui on été stockés avant de faire appel à realloc().

Par conséquent, si on veut faire la même opération avec seulement les instructions malloc() et free(), c'est plus compliqué et cela donne :

Code : C

```
char *maChaine = malloc(15 * sizeof(char)), *copieDeMaChaine = NULL;
/* 15 ou plus probablement la valeur d'une variable. */
int i;

if (maChaine == NULL)
    pb_memoire();
for (i=0; i<15; ++i)
    maChaine[i] = 0; /* On met tout à zéro, ce que fait calloc().
quelques instructions */

copieDeMaChaine = malloc(15 * sizeof(char));
if (copieDeMaChaine == NULL)
    pb_memoire();

for (i = 0; i < 15; i++)
    copieDeMaChaine[i] = maChaine[i]; /* On enregistre les données
pour ne pas les perdre. */

free(maChaine); /* On libére la zone mémoire avant d'en acquérir
une nouvelle. */
maChaine = malloc(20 * sizeof(char)); /* On demande plus de place
en mémoire. */

if (maChaine == NULL)
    pb_memoire();

for (i = 0; i < 15; i++)
    maChaine[i] = copieDeMaChaine[i]; /* On récupère les valeurs
qu'on a sauvegardées. */
free(copieDeMaChaine); /* On peut maintenant supprimer les données
sauvegardées.
```

```
quelques instructions */
free(maChaine); /* On n'oublie pas de libérer la mémoire à la fin
du programme ou de la fonction. */
```

J'ai parlé de concision pour calloc() tout à l'heure. Là, on est servi !

Dans ces exemples, on pourrait remplacer la boucle for() par la fonction strcpy(). Cela est vrai parce qu'on utilise des char mais on ne peut plus utiliser cette fonction dès lors qu'on manipule autre chose que des char, comme des int, des double etc. Donc l'utilisation de la boucle for() est une technique qui marchera dans tous les cas de figure.

La bonne utilisation de realloc

L'instruction `ptr = realloc(ptr, nbElements * sizeof(typeElement));` présente un défaut : Avant l'appel de realloc, le pointeur ptr pointe sur une zone valide de mémoire si on a appelé malloc ou calloc avant. Si la ré-allocation échoue, realloc renvoie NULL, qui est affecté à ptr, or realloc préserve le bloc mémoire précédemment alloué, ce qui signifie que le bloc mémoire n'est pas libéré. En affectant NULL à ptr, on perd donc l'adresse de la zone mémoire allouée !

La solution consiste à créer deux pointeurs : ptr et ptr_realloc par exemple, et c'est à ptr_realloc qu'on affectera le résultat de realloc. Il suffit ensuite de tester sa valeur, et si elle n'est pas nulle, on la réaffecte à ptr.

Créons la fonction de réallocation sécurisée suivante :

Code : C

```
void* realloc_s (void **ptr, size_t taille)
{
 void *ptr_realloc = realloc(*ptr, taille);

 if (ptr_realloc != NULL)
     *ptr = ptr_realloc;
/* Même si ptr_realloc est nul, on ne vide pas la mémoire. On
laisse l'initiative au programmeur. */

 return ptr_realloc;
}
```

Un exemple concret : réallocation sécurisée

Dans l'exemple suivant, j'ai créé un programme qui demande à l'utilisateur de rentrer une chaîne de caractère et un caractère. Dans cette chaîne de caractère, on va "supprimer" un caractère particulier. En réalité, on crée simplement un décalage en écrasant le caractère à supprimer par le suivant. La fonction deleteCharInString ne gère pas la mémoire. Il faut donc réajuster la taille de la mémoire pour correspondre parfaitement à la nouvelle vraie taille.

Code : C - exemple.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define OS /* définir ici WINDOWS, LINUX ou MAC-OSX */
#define PAUSE puts("Appuyez sur entree pour continuer..."); \
viderBuffer(); /* En effet, viderBuffer permet également de faire
une pause dans le programme ! */

char* deleteCharInString(char c, char *string);
void* realloc_s(void **ptr, size_t taille); /* déclaration de la
```

```
fonction de réallocation sécurisée */
size_t rentrerChaine(char **tailleChaine); /* renvoie la taille de
la chaine */
char rentrerCaractere(void);
void viderBuffer(void);

int main(int argc, char *argv[])
{
    char *chaine, toDelete;
    size_t tailleChaine;

    puts("Entrez une chaine de caractere :");
    tailleChaine = rentrerChaine(&chaine);

    if (tailleChaine > 0) {
        puts("Entrez le caractere de la chaine a supprimer :");
        toDelete = rentrerCaractere();

        printf("\nVous avez entre la chaine \"%s\" et le caractere
'%c'.\n", chaine, toDelete);
    /* On affiche à l'écran ce que l'utilisateur a rentré
(éventuellement tronqué si c'était trop grand pour contenir
dans le tableau). */

        deleteCharInString(toDelete, chaine);
realloc s(&chaine, (strlen(chaine)+1) * sizeof(char)); /* La taille
de la chaine a sûrement changé :
on ajuste la taille mémoire bien que ça ne soit pas indispensable,
mais cela permet ici de réduire l'occupation en mémoire
de la chaîne. */

        printf("\nLa chaine apres suppression de '%c' vaut
maintenant :\n%s\n", toDelete, chaine);
    }
    free(chaine);

#ifdef WINDOWS
    /*
* Pause pour pouvoir lire avant que la console ne disparaisse.
* Ceci est une version portable.
*/
     PAUSE
#endif

    return EXIT_SUCCESS;
}

char* deleteCharInString(char c, char *string)
{
    size_t cpt = 0, count = 0;

    if (c != '\0') { /* si on ne cherche pas à supprimer le
caractère de fin de chaîne */
        while (string[cpt] != c && string[cpt] != '\0')
            ++cpt;
/* On se positionne sur la première occurence du caractère à
supprimer ou à la fin si le caractère à supprimer n'est pas dans la
chaîne. */
        if (string[cpt] == c) {
            do ++count;
            while (string[cpt+count] == c);
            string[cpt] = string[cpt+count];
            /* On remplace le caractère à supprimer par le suivant.
*/
            while (string[cpt+count] != '\0') {
            /* si on n'est pas à la fin de la chaîne, il faut
recommencer.
On fait la même opération pour le caractère suivant. */
                ++cpt;
                while (string[cpt+count] == c)
```

```
                    ++count;
                string[cpt] = string[cpt+count];
            /* On affecte le prochain caractère qui ne soit pas à
détruire. */
                }
            }
        }
        return string;
}

void* realloc_s(void **ptr, size_t taille)
{
void *ptr_realloc = realloc(*ptr, taille);

     if (ptr_realloc != NULL)
         *ptr = ptr_realloc;

     return ptr_realloc;
}

void viderBuffer(void)
{
    char poubelle;

    do poubelle = getchar();
    while (poubelle != '\n' && poubelle != EOF);
}

size_t rentrerChaine(char **chaine) /* alloue dynamiquement de la
mémoire pour chaque caractère rentré : pas de gaspillage de
mémoire. */
{
     size_t tailleChaine = 1;

     *chaine = malloc(sizeof(char));

     if (*chaine != NULL) {
         **chaine = getchar();
         while (*chaine != NULL && (*chaine)[tailleChaine-1] != '\n'
&& (*chaine)[tailleChaine-1] != EOF)
if (realloc_s(chaine, ++tailleChaine * sizeof(char)))
                 (*chaine)[tailleChaine-1] = getchar();
             else {
                 free(*chaine); /* la mémoire est pleine : on la
libère. */
                 *chaine = NULL;
                 viderBuffer();
                 tailleChaine = 1;
             }
         if (*chaine != NULL)
             (*chaine)[tailleChaine-1] = '\0';
     }

     return --tailleChaine;
}

char rentrerCaractere(void)
{
    char input = getchar();

    while (input == '\n') {
        puts("Vous n'avez pas entre un caractere valide. Veuillez
recommencer.");
 input = getchar(); /* prend un nouveau caractere */
    }
    viderBuffer();

    return input;
}
```

Ceci produit :

Code : Console

```
Entrez une chaine de caractere :
Ceci est une chaine de caractere quelconque. On va en supprimer tous les 'e'.
Entrez le caractere de la chaine a supprimer :
e

Vous avez entre la chaine "Ceci est une chaine de caractere quelconque. On va en su

La chaine apres suppression de 'e' vaut maintenant :
Cci st un chain d caractr qulconqu. On va n supprimr tous ls ''.
```

Applications pour le TP du pendu

Dans le TP du pendu, j'utilise **calloc()** pour allouer et initialiser la mémoire nécessaire au tableau de int (lettresTrouvees, déclaré à la ligne 16) de même longueur que le mot à trouver et dont les éléments passent à 1 lorsque l'utilisateur a trouvé une des lettres du mot.

Code : C

```
#include <stdio.h>
#include <stdlib.h>
#include "dico.h"
#include "fonctions_pendu.h"

#define COUPS_RESTANTS 7

typedef unsigned short int entier;

int main(int argc, char* argv[])
{

 char lettre, motSecret[32], *lettresProposees, *recherche;

 entier coupsRestants, nombreLettresProposees, cpt, tailleMot,
   *lettresTrouvees; /* Un tableau de booléens. Chaque case
correspond à une lettre du mot secret. 0 = lettre non trouvée, 1 =
lettre trouvée */

 enum {EN_COURS, GAGNEE, PERDUE} partie;

 do {

/**************************INITIALISATION DES
VARIABLES************************/
     putchar('\n');
     if (piocherMot(motSecret) == 0)
         pb_memoire();

     tailleMot = strlen(motSecret);

lettresTrouvees = calloc(tailleMot, sizeof(entier)); /* Le tableau
ne comporte que des zéros ! */
     if (lettresTrouvees == NULL) /* test de la validité du
pointeur */
         pb_memoire();

     partie = EN_COURS;

     nombreLettresProposees = 0;

     coupsRestants = COUPS_RESTANTS;
```

```
    coupsRestants = COUPS_RESTANTS;
/*************************INITIALISATION DES
VARIABLES************************/

 /* corps du programme */

 } while(rejouer()); /* une boucle do while parce qu'on joue au
moins une fois
et on demande à l'utilisateur s'il veut ou non rejouer. */

 return 0;
}
```

Ici, l'allocation et initialisation du pointeur lettresTrouvees (qui comme son nom ne l'indique pas est un pointeur de entier (unsigned short int)) se fait à la ligne 29. C'est quand même plus commode ainsi qu'utiliser malloc() et une boucle for().

En ce qui concerne **realloc()**, je m'en sers pour que le programme mémorise chaque nouvelle lettre que l'utilisateur propose afin que lorsque celui-ci propose une lettre qu'il a déjà proposée, on affiche un message et on ne lui enlève pas un coup d'essai. Pour cela, il faut allouer dynamiquement de la mémoire. Sauf qu'au premier coup, aucune case mémoire n'est encore allouée. Je fais donc appel à **malloc()**. Par la suite, je veux augmenter d'une case cette zone mémoire pour y stocker la nouvelle lettre. Je fais donc appel à **realloc_s()**.

Code : C

```
lettre = proposer(lettresProposees, nombreLettresProposees);

if (nombreLettresProposees == 0) {
    lettresProposees = malloc(sizeof(char));
    if (lettresProposees == NULL)
        pb_memoire();
}
else if(realloc s(&lettresProposees, (nombreLettresProposees+1) *
sizeof(char))) /* appel à la fonction sécurisée */
     lettresProposees[nombreLettresProposees++] = lettre;
else {
/* gestion erreur */
     free(lettresProposees);
     free(lettresTrouvees);
     puts("ERREUR : memoire saturee");
     exit(EXIT_FAILURE);
}
```

Q.C.M.

Le premier QCM de ce cours vous est offert en libre accès.
Pour accéder aux suivants

Connectez-vous Inscrivez-vous

La fonction malloc() admet :

- ◯ 1 argument
- ◯ 2 arguments
- ◯ 3 arguments

La fonction calloc() admet :

- ◯ 1 argument
- ◯ 2 arguments
- ◯ 3 arguments

La fonction realloc() admet :

- ◯ 1 argument
- ◯ 2 arguments
- ◯ 3 arguments

Dans le code suivant, la variable ptr_int est de type `int*` et a été correctement initialisé. La variable taille a été correctement déclaré (`size_t` ou `int`) et initialisé.
L'instruction suivante illustre une bonne utilisation de realloc() :

Code : C

```
ptr_int = realloc(ptr_int, taille * sizeof(int));
```

- ◯ faux
- ◯ vrai

Correction !

Statistiques de réponses au QCM

Qu'une initialisation soit requise ou non, je vous conseille d'utiliser la fonction **calloc()**, sauf s'il ne faut de la place que pour un seul élément et qu'il n'a pas besoin d'être initialisé à 0. En effet, il est plus court d'écrire : `monPointeur = malloc(sizeof(int));` que d'écrire : `monPointeur = calloc(1, sizeof(int));`

Bien souvent, des valeurs doivent être initialisées à zéro, ce que fait pour nous calloc() ! Donc autant se servir de cette fonctionnalité.

La fonction **realloc()** est quant à elle d'une grande commodité. Elle est utilisée dans la plupart des programmes C gérant dynamiquement la mémoire. Utilisez cependant la version sécurisée (l'avantage de le mettre sous forme de fonction est de ne pas déclarer ptr_realloc dans le corps du programme).

Ceci étant, il ne faut pas oublier d'utiliser la fonction **free()** pour libérer la mémoire, dans tous les cas.
Je vous conseille d'ailleurs d'écrire une fonction *finalize()* ou *delete_all()* qui fera appel à free() pour libérer proprement la mémoire des pointeurs que vous aurez à utiliser, pour être sûr de ne pas provoquer de fuite mémoire.

Voilà, c'est fini pour ce tuto. A présent, vous avez tous les outils pour produire des algorithmes plus courts qui gèreront efficacement la mémoire de votre ordinateur.

Allez, amusez-vous bien !