

Les chaînes de caractères en C

par Nicolas Joseph ([home](#)) ([Blog](#))

Date de publication : 14 Novembre 2005

Dernière mise à jour : 1er Août 2007

Après un rappel sur la structure des chaînes de caractères en C et un aperçu des fonctions de la bibliothèque standard servant à manipuler ces chaînes, je vous propose de créer notre propre bibliothèque de manipulation des chaînes de caractères.

- I - Les chaînes de caractères en C
- II - Les fonctions de la bibliothèque standard
 - II-A - strcpy
 - II-B - strncpy
 - II-C - strcat
 - II-D - strncat
 - II-E - strcmp
 - II-F - strncmp
 - II-G - strlen
 - II-H - strchr
 - II-I - strrchr
 - II-J - strstr
 - II-K - strspn
 - II-L - strcspn
 - II-M - strpbrk
 - II-N - strtok
- III - Création de fonctions avancées
 - III-A - Modifier la case d'une chaîne de caractère
 - III-B - Connaître l'index d'une sous-chaîne
 - III-C - Extraire une sous-chaîne
 - III-D - Découper une chaîne
 - III-E - Fusionner plusieurs chaînes de caractères
 - III-F - Remplacer une partie d'une chaîne
 - III-G - Eliminer les espaces superflus
- IV - Remerciements
- V - Code source complet

I - Les chaînes de caractères en C

En C, il n'existe pas de type de variable pour les chaînes de caractères comme il en existe pour les entiers (**int**) ou pour les caractères (**char**). Les chaînes de caractères sont en fait stockées dans un tableau de caractères dont la fin est marquée par un caractère nul, de code ASCII 0 et représenté par le caractère **\0** pour ne pas confondre avec le caractère **0** (zéro) qui à un code ASCII de 48. En mémoire la chaîne "Bonjour" est représentée ainsi :

```
B|o|n|j|o|u|r|\0
```

Tout ce qui suit le caractère **\0** sera ignoré :

```
char s[14] = "Hello\0World!";  
printf ("%s\n", s);
```

Affichera seulement "Hello". Il ne faut donc pas oublier de réserver une place supplémentaire pour le caractère de fin de chaîne sinon on obtient un simple tableau de caractères et dans ce cas, son utilisation en tant que chaîne de caractère mène à un comportement indéfini.

Il existe plusieurs manières de déclarer et d'initialiser une chaîne de caractères :

```
char s1[11] = "Developpez";  
char s2[] = "Developpez";  
char *s3 = "Developpez";
```

La première méthode réserve une zone de 11 bytes et la chaîne "Developpez" y est stockée. La seconde méthode laisse le compilateur calculer la taille appropriée (11 bytes dans notre cas). La dernière méthode ne réserve pas de mémoire, elle se contente de stocker l'adresse d'une chaîne de caractères qui peut se trouver dans un endroit de la mémoire non modifiable par le programme, dans ce cas toute modification de la chaîne s3 se conclura par une erreur de segmentation. il est donc conseillé d'écrire :

```
const char *s3 = "Developpez";
```

Déclarer une chaîne de caractères constante est aussi une bonne habitude à prendre lors du passage d'une chaîne comme paramètre d'une fonction qui ne modifie pas cette dernière.

Si vous souhaitez utiliser une chaîne de taille variable, il faut utiliser les mécanismes d'allocation dynamique de la mémoire :

```
char *s = malloc (sizeof (*s) * 256);  
/*  
 * Utilisation d'une chaîne de 255 caractères maximum  
 */  
free (s);
```

II - Les fonctions de la bibliothèque standard

Voici une courte description des fonctions de la bibliothèque standard destinée à la manipulation des chaînes de caractères. Il ne faut pas oublier d'inclure *string.h* avant de les utiliser.

II-A - strcpy


```
char *strcpy (char *s, const char *ct);
```

Copie la chaîne *ct* dans *s* (y compris le caractère de fin de chaîne), retourne *s*.

II-B - strncpy

```
char *strncpy (char *dst, const char *src, size_t n);
```

Identique à [strcpy](#) en se limitant à *n* caractères.

 *Les fonctions limitant le nombre de caractères sont une bonne habitude à prendre pour éviter, dans le cas de [strcpy](#), de copier une chaîne trop grande et entraîner un comportement indéfini. Cependant, il faut tout de même faire attention puisque, toujours avec [strcpy](#), si l'on veut copier une chaîne trop grande, le caractère de fin de chaîne de ne sera pas copié : on obtient une chaîne invalide. Pour faire une utilisation sécurisée de ces fonctions, il faut ajouter nous-même un caractère de fin de chaîne à la fin du tableau.*

II-C - strcat

```
char *strcat (char *s, const char *ct);
```

Concatène la chaîne *ct* à la suite de *s* et retourne *s*

II-D - strncat

```
char *strncat (char *s, const char *ct, size_t n);
```

Identique à [strcat](#) en se limitant à *n* caractères.

II-E - strcmp

```
int strcmp (const char *cs, const char *ct);
```

Compare les chaînes *cs* et *ct* et renvoi :

- Une valeur négative si, lexicalement, $cs < ct$
- Une valeur nulle si $cs == ct$
- Une valeur positive si $cs > ct$

II-F - strncmp

```
int strncmp (const char *cs, const char* ct, size_t n);
```

Identique à [strcmp](#) en se limitant à n caractères.

II-G - strlen

```
size_t strlen (const char *cs);
```

Retourne le nombre de caractères de cs sans tenir compte du caractère de fin de chaîne.

II-H - strchr

```
char *strchr (const char *cs, int c);
```

Retourne l'adresse de la première occurrence du caractère c dans la chaîne cs en partant du début de la chaîne.

II-I - strrchr

```
char *strrchr (const char *cs, int c);
```

Identique à [strchr](#) en commençant par la fin de cs .

II-J - strstr

```
char *strstr (const char *cs, const char *ct);
```

Identique à [strchr](#) sauf qu'il s'agit de la première occurrence de la sous-chaîne ct .

II-K - strspn

```
size_t strspn (const char *cs, const char *ct);
```

Cette fonction retourne le nombre de caractères du début de cs appartenant à ct .

II-L - strcspn

```
size_t strcspn (const char *cs, const char *ct);
```

C'est l'inverse de la fonction [strspn](#), cette fois, les caractères ne font pas partie de l'ensemble ct .

II-M - strpbrk

```
char *strpbrk (const char *cs, const char *ct);
```


Identique à [strchr](#) sauf que cette fois, c'est l'ensemble des caractères de *ct* qui est recherché.

II-N - strtok

```
char *strtok (char *s, const char *t);
```

Cette fonction décompose la chaîne *s* en sous-chaîne délimitées par un caractère appartenant à *ct*. Un appel ultérieur à [strtok](#) avec *s* égale à NULL, retourne la sous-chaîne suivante ; *ct* peut être différent à chaque appel.

La chaîne passée en argument est modifiée par la fonction : elle ajoute un caractère de fin de chaîne à la place du séparateur.

 *Le fichier [string.h](#) contient le prototype d'autres fonctions mais qui ne sont pas utiles pour la manipulation des chaînes de caractères.*

III - Création de fonctions avancées

Notre but étant de créer une bibliothèque, les fonctions publiques devront commencer par un préfixe pour éviter les conflits de noms. Le préfixe *str* suivie d'une lettre est réservé pour les fonctions de la bibliothèque standard, nous utiliserons donc *str_*. Nous ne devons pas faire de suppositions hasardeuses sur les paramètres fournis par l'utilisateur de notre bibliothèque, aussi vérifier la validité de ceux-ci est nécessaires pour éviter tout arrêt brutal du programme.

III-A - Modifier la case d'une chaîne de caractère

Le fichier d'en tête *ctype.h* propose les fonctions **tolower** et **toupper** pour mettre un caractère respectivement en minuscule et en majuscule, il est intéressant de proposer la même chose mais pour une chaîne de caractères :

```
char *str_tolower (const char *ct)
{
    char *s = NULL;

    if (ct != NULL)
    {
        int i;


        /* (1) */
        s = malloc (sizeof (*s) * (strlen (ct) + 1));
        if (s != NULL)
        {
            /* (2) */
            for (i = 0; ct[i]; i++)
            {
                s[i] = tolower (ct[i]);
            }
            s[i] = '\0';
        }
    }
    return s;
}

char *str_toupper (const char *ct)
{
    char *s = NULL;

    if (ct != NULL)
    {
        int i;

        /* (1) */
        s = malloc (sizeof (*s) * (strlen (ct) + 1));
        if (s != NULL)
        {
            /* (2) */
            for (i = 0; ct[i]; i++)
            {
                s[i] = toupper (ct[i]);
            }
            s[i] = '\0';
        }
    }
    return s;
}
```

Rien de bien compliqué, on commence par allouer une nouvelle chaîne de même taille (1) puis on recopie chaque caractère un par un en modifiant sa case (2).

 Dans cet exemple, pour parcourir la chaîne de caractère j'ai utilisé une boucle **for** pour des questions de lisibilité, mais on rencontre souvent ce genre de code :

```
while (*s++)
{
  /* ... */
}
```

La notation `*s` est équivalente à `s[0]` donc permet de retourner le premier caractère d'une chaîne. Le parcours de la chaîne se fait simplement en incrémentant l'adresse du tableau. La boucle s'arrête lorsque l'on atteint le caractère de fin de chaîne qui vaut 0 (donc faux). Attention, avec cette méthode, on perd l'adresse du premier élément (en cas d'allocation dynamique, impossible de libérer la mémoire) cependant en cas de restriction d'espace mémoire, cela permet d'économiser une variable.


III-B - Connaître l'index d'une sous-chaîne

Il existe la fonction **strstr** qui permet de trouver l'adresse d'une sous-chaîne mais je trouve plus intéressant de connaître l'index de celle-ci dans le tableau :

```
int str_istr (const char *cs, const char *ct)
{
  int index = -1;

  if (cs != NULL && ct != NULL)
  {
    char *ptr_pos = NULL;

    ptr_pos = strstr (cs, ct);
    if (ptr_pos != NULL)
    {
      index = ptr_pos - cs;
    }
  }
  return index;
}
```

 Il faut vérifier le retour de la fonction car si la sous-chaîne n'est pas trouvée, l'indice vaut -1 ce qui provoquera un comportement indéfini en cas d'utilisation de l'indice dans un tableau.

III-C - Extraire une sous-chaîne

Cette fonction permet d'extraire une sous-chaîne de `s` comprise entre l'indice `start` et `end`.

```
char *str_sub (const char *s, unsigned int start, unsigned int end)
{
  char *new_s = NULL;

  if (s != NULL && start < end)
  {
    /* (1)*/
    new_s = malloc (sizeof (*new_s) * (end - start + 2));
    if (new_s != NULL)
    {
      int i;

      /* (2) */
      for (i = start; i <= end; i++)
```



```
/* (3) */
    {
        new_s[i-start] = s[i];
    }
}
else
{
    fprintf (stderr, "Memoire insuffisante\n");
    exit (EXIT_FAILURE);
}
}
return new_s;
}
```

Grâce aux indices, on calcule la taille de la sous-chaîne (1), ensuite on parcourt s entre les deux indices (2) pour copier chaque caractère dans la sous-chaîne (3).

III-D - Découper une chaîne

J'ai découvert cette fonction en programmant en **Perl**, elle permet de découper une chaîne de caractère suivant un délimiteur et de placer chaque sous-chaîne dans un tableau terminé par NULL.


```
char **str_split (char *s, const char *ct)
{
    char **tab = NULL;

    if (s != NULL && ct != NULL)
    {
        int i;
        char *cs = NULL;
        size_t size = 1;

        /* (1) */
        for (i = 0; (cs = strtok (s, ct)); i++)
        {
            if (size <= i + 1)
            {
                void *tmp = NULL;

                /* (2) */
                size <<= 1;
                tmp = realloc (tab, sizeof (*tab) * size);
                if (tmp != NULL)
                {
                    tab = tmp;
                }
                else
                {
                    fprintf (stderr, "Memoire insuffisante\n");
                    free (tab);
                    tab = NULL;
                    exit (EXIT_FAILURE);
                }
            }
        }
        /* (3) */
        tab[i] = cs;
        s = NULL;
    }
    tab[i] = NULL;
    return tab;
}
```

Tant que `strtok` nous renvoi une adresse non nulle (1), on augmente la taille du tableau d'une case (2) dans laquelle on stocke l'adresse retournée par `strtok` (3).

 *La variable `s` passée en paramètre est modifiée par la fonction `strtok`. De plus le tableau de pointeurs renvoyé par notre fonction fait référence à la chaîne passée en paramètre, par conséquent, elle ne doit pas être modifiée ni détruite si vous utilisez le tableau de sous-chaînes.*

III-E - Fusionner plusieurs chaînes de caractères

Encore une fonction inspirée de **Perl**! Cette fois ci, il s'agit de faire l'inverse de la fonction `str_split` en réunissant des chaînes de caractères grâce à un séparateur.

```
char *str_join (char *cs, ...)
{
    va_list va;
    const char *ct;
    char *s = NULL;
    size_t size = 0;

    va_start (va, cs);
    /* (1) */
    while ((ct = va_arg (va, char *)) != NULL)
    {
        void *tmp = NULL;

        /* (2) */
        size += strlen (ct) + strlen (cs);
        tmp = realloc (s, sizeof (*s) * (size + 1));
        if (tmp != NULL)
        {
            if (s == NULL)
            {
                /* (3) */
                s = tmp;
                strcpy (s, ct);
            }
            else
            {
                /* (4) */
                s = tmp;
                strcat (s, cs);
                strcat (s, ct);
            }
        }
        else
        {
            fprintf (stderr, "Memoire insuffisante\n");
            free (s);
            s = NULL;
            exit (EXIT_FAILURE);
        }
    }
    return s;
}
```

Pour pouvoir gérer un nombre variable d'arguments, la liste doit être terminée par la valeur `NULL` (1). Ensuite pour chaque argument, on calcule la nouvelle taille de la chaîne finale sans oublier d'ajouter la taille du séparateur (2). Pour la première chaîne, on la copie simplement (3) et pour les autres (4), on commence par concaténer le séparateur puis la nouvelle chaîne.

III-F - Remplacer une partie d'une chaîne

Cette fonction permet de remplacer *length* caractères de la chaîne *s* à partir de l'indice *start* par une nouvelle chaîne *ct*. Une nouvelle chaîne est créée, il ne faut pas oublier de libérer la mémoire.

```
char *str_replace (const char *s, unsigned int start, unsigned int length, const char *ct)
{
    char *new_s = NULL;

    if (s != NULL && ct != NULL && start >= 0 && length > 0)
    {
        size_t size = strlen (s);

        /* (1) */
        new_s = malloc (sizeof (*new_s) * (size - length + strlen (ct) + 1));
        if (new_s != NULL)
        {
            /* (2) */
            memcpy (new_s, s, start);
            /* (3) */
            memcpy (&new_s[start], ct, strlen (ct));
            /* (4) */
            memcpy (&new_s[start + strlen (ct)], &s[start + length], size - length - start + 1);
        }
    }
    else
    {
        fprintf (stderr, "Memoire insuffisante\n");
        exit (EXIT_FAILURE);
    }
    return new_s;
}
```

- 1 On commence par créer un nouvel emplacement mémoire pour stocker la chaîne modifiée
- 2 Ensuite, la nouvelle chaîne est créée par copier/coller du début de *s*
- 3 Puis à l'indice *start*, on ajoute la chaîne de remplacement *ct*
- 4 Pour finir par la fin de la chaîne de départ *s* (sans oublier de copier le caractère de fin de chaîne).

III-G - Eliminer les espaces superflus

Pour finir notre bibliothèque, je vous propose une fonction qui supprime les espaces superflus dans une chaîne de caractère.

```
char *str_strip (const char *string)
{
    char *strip = NULL;

    if (string != NULL)
    {
        strip = malloc (sizeof (*strip) * (strlen (string) + 1));
        if (strip != NULL)
        {
            int i, j;
            int ps = 0;

            for (i = 0, j = 0; string[i]; i++)
            {
                if (string[i] == ' ')
                {
                    if (ps == NULL)
                    {
```

```
        strip[j] = string[i];
        ps = 1;
        j++;
    }
    else
    {
        strip[j] = string[i];
        ps = 0;
        j++;
    }
}
else
{
    fprintf (stderr, "Memoire insuffisante\n");
    exit (EXIT_FAILURE);
}
}
return strip;
}
```

Pour supprimer les espaces, il suffit de recopier la chaîne de caractère initiale et lorsque l'on rencontre un espace, on regarde s'il s'agit du premier (si l'on a déjà rencontré un espace, la variable *ps* vaut 1), si c'est le cas, on le copie comme n'importe quel caractère et l'on met *ps* à 1, par contre si ce n'est pas le premier, on ne le copie pas. Bien sûr, si l'on copie autre chose qu'un espace, il ne faut pas oublier de mettre *ps* à zéro pour que le prochain espace soit copié.

IV - Remerciements

Merci à khayyam90 pour la relecture attentive de cet article.

V - Code source complet

En plus des fonctions vues dans la partie précédente, le fichier *str.h* redéfinit les fonctions de la bibliothèque standard, ceci peut nous permettre par la suite de réécrire ses fonctions (pour ajouter la vérification de la validité des paramètres par exemple).

Téléchargez l'[archive zippée](#).

```
str.h
#ifndef H_STRING
#define H_STRING

#include <string.h>

/* Redefinition des fonctions standards pour un usage futur, par exemple verifier
   la validite des arguments. */
#define str_cpy strcpy
#define str_ncpy strncpy
#define str_cat strcat
#define str_ncat strncat
#define str_cmp strcmp
#define strncmp strncmp
#define str_len strlen
#define str_chr strchr
#define str_rchr strrchr
#define str_str strstr
#define str_spn strspn
#define str_pbrk strpbrk
#define str_tok strtok

char *str_tolower (const char *);
char *str_toupper (const char *);
int str_istr (const char *, const char *);
char *str_sub (const char *, unsigned int, unsigned int);
char **str_split (char *, const char *);
char *str_join (char *, ...);
char *str_replace (const char *, unsigned int, unsigned int, const char *);
char *str_strip (const char *);

#endif /* not H_STRING */
```

```
str.c
#include <ctype.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include "sstring.h"

char *str_tolower (const char *ct)
{
    char *s = NULL;

    if (ct != NULL)
    {
        int i;

        s = malloc (sizeof (*s) * (strlen (ct) + 1));
        if (s != NULL)
        {
            for (i = 0; ct[i]; i++)
            {
                s[i] = tolower (ct[i]);
            }
        }
    }
}
```

str.c

```
        s[i] = '\0';
    }
}
return s;
}

char *str_toupper (const char *ct)
{
    char *s = NULL;

    if (ct != NULL)
    {
        int i;

        s = malloc (sizeof (*s) * (strlen (ct) + 1));
        if (s != NULL)
        {
            for (i = 0; ct[i]; i++)
            {
                s[i] = toupper (ct[i]);
            }
            s[i] = '\0';
        }
    }
    return s;
}

int str_istr (const char *cs, const char *ct)
{
    int index = -1;

    if (cs != NULL && ct != NULL)
    {
        char *ptr_pos = NULL;

        ptr_pos = strstr (cs, ct);
        if (ptr_pos != NULL)
        {
            index = ptr_pos - cs;
        }
    }
    return index;
}

char *str_sub (const char *s, unsigned int start, unsigned int end)
{
    char *new_s = NULL;

    if (s != NULL && start < end)
    {
        new_s = malloc (sizeof (*new_s) * (end - start + 2));
        if (new_s != NULL)
        {
            int i;

            for (i = start; i <= end; i++)
            {
                new_s[i-start] = s[i];
            }
        }
        else
        {
            fprintf (stderr, "Memoire insuffisante\n");
            exit (EXIT_FAILURE);
        }
    }
    return new_s;
}
```

str.c

```
}

char **str_split (char *s, const char *ct)
{
    char **tab = NULL;

    if (s != NULL && ct != NULL)
    {
        int i;
        char *cs = NULL;
        size_t size = 1;

        for (i = 0; (cs = strtok (s, ct)); i++)
        {
            if (size <= i + 1)
            {
                void *tmp = NULL;

                size <<= 1;
                tmp = realloc (tab, sizeof (*tab) * size);
                if (tmp != NULL)
                {
                    tab = tmp;
                }
                else
                {
                    fprintf (stderr, "Memoire insuffisante\n");
                    free (tab);
                    tab = NULL;
                    exit (EXIT_FAILURE);
                }
            }
            tab[i] = cs;
            s = NULL;
        }
        tab[i] = NULL;
    }
    return tab;
}

char *str_join (char *cs, ...)
{
    va_list va;
    const char *ct;
    char *s = NULL;
    size_t size = 0;

    va_start (va, cs);
    while ((ct = va_arg (va, char *)) != NULL)
    {
        void *tmp = NULL;

        size += strlen (ct) + strlen (cs);
        tmp = realloc (s, sizeof (*s) * (size + 1));
        if (tmp != NULL)
        {
            if (s == NULL)
            {
                s = tmp;
                strcpy (s, ct);
            }
            else
            {
                s = tmp;
                strcat (s, cs);
                strcat (s, ct);
            }
        }
    }
}
```


str.c

```
    }
    else
    {
        fprintf (stderr, "Memoire insuffisante\n");
        free (s);
        s = NULL;
        exit (EXIT_FAILURE);
    }
}
return s;
}

char *str_replace (const char *s, unsigned int start, unsigned int lenght, const char *ct)
{
    char *new_s = NULL;

    if (s != NULL && ct != NULL && start >= 0 && lenght > 0)
    {
        size_t size = strlen (s);

        new_s = malloc (sizeof (*new_s) * (size - lenght + strlen (ct) + 1));
        if (new_s != NULL)
        {
            memcpy (new_s, s, start);
            memcpy (&new_s[start], ct, strlen (ct));
            memcpy (&new_s[start + strlen (ct)], &s[start + lenght], size - lenght - start + 1);
        }
    }
    else
    {
        fprintf (stderr, "Memoire insuffisante\n");
        exit (EXIT_FAILURE);
    }
    return new_s;
}

char *str_strip (const char *string)
{
    char *strip = NULL;

    if (string != NULL)
    {
        strip = malloc (sizeof (*strip) * (strlen (string) + 1));
        if (strip != NULL)
        {
            int i, j;
            int ps = 0;

            for (i = 0, j = 0; string[i]; i++)
            {
                if (string[i] == ' ')
                {
                    if (ps == NULL)
                    {
                        strip[j] = string[i];
                        ps = 1;
                        j++;
                    }
                }
                else
                {
                    strip[j] = string[i];
                    ps = 0;
                    j++;
                }
            }
        }
    }
}
```

str.c

```
    else
    {
        fprintf (stderr, "Memoire insuffisante\n");
        exit (EXIT_FAILURE);
    }
}
return strip;
}
```

