

# Rappels sur les chaînes de caractères en C

En langage C, une chaîne de caractères est un tableau de caractères (chaque élément du tableau = un caractère, sur un octet), terminé par un dernier élément particulier dit « caractère de fin de chaîne » ou « caractère nul » ( 0 binaire, souvent noté '\0'). Quand on donne la longueur de la chaîne, on n'inclut jamais le '\0' de fin (mais il faut en tenir compte dans la taille en mémoire de la chaîne !).

En C, une variable de type « tableau » contient l'adresse mémoire de son premier élément. C'est donc aussi le cas d'un variable de type « chaîne de caractères ».

Ex :

La chaîne « hello » est stockée en mémoire comme suit :

adresse de début→	h
	e
	l
	l
	o
	\0

Sa longueur ( fonction `strlen`, cf. § 3) vaut 5, mais elle occupe 6 octets en mémoire.

## 1. Déclaration d'une chaîne de caractères

Comme pour les tableaux, il existe deux façons de déclarer une chaîne de caractères, par exemple :

```
char *chaine; // chaine est un pointeur auquel vous devrez donner
              //une valeur par un malloc (ou par une initialisation)
```

ou

```
char chaine[10]; //10 est la taille réservée en mémoire pour chaine
```

Cette seconde façon est la plus simple et elle est recommandée, sauf lorsqu'il sera nécessaire de créer des chaînes dynamiquement (de longueur non prédéfinie).

Par exemple, si on vous demande de déclarer une chaîne de longueur maximale prédéfinie 100 caractères, un débutant en programmation C écrira souvent :

```
char chaine[100];
```

Cette déclaration n'est pas très bonne pour deux raisons :

- si on veut effectivement mettre 100 caractères dans la variable `chaine`, elle doit faire 101 caractères de long : il faut en effet prévoir une place pour le caractère de fin de chaîne ('\0').
- Il faut éviter de mettre des constantes « en dur » (ici 100) dans un programme, il vaut mieux déclarer une constante pour cela. En C, ça se fait typiquement à l'aide d'un `#define` (directive du préprocesseur).

En résumé, il vaut mieux écrire :

```
#define LONGUEUR_MAX_CHAINE 100
```

...

```
char chaine[LONGUEUR_MAX_CHAINE+1];
```

### **Cas particulier : déclaration avec initialisation**

Comme pour tous les types de variable en C, on peut initialiser une variable de type « chaîne de caractères » avec une chaîne dite « constante ». Ex :

```
char chaine[LONGUEUR_MAX_CHAINE+1] = "hello";
```

**Attention !** : c'est à vous de vérifier que la chaîne constante ne soit pas trop grande : sa longueur doit être inférieure ou égale à `LONGUEUR_MAX_CHAINE`. La plupart des compilateurs C ne font pas ce genre de vérification.

**Remarque 1** : une chaîne constante est automatiquement terminée par le caractère de fin de chaîne (`'\0'`). Vous n'avez pas à le rajouter.

**Remarque 2** : ce qui est possible à l'initialisation ne sera pas possible ensuite (cf. § 4) : l'instruction `chaine = "hello";` sera refusée par le compilateur.

## 2. Saisie et affichage d'une chaîne de caractères

La saisie d'une chaîne de caractères tapée au clavier peut se faire avec les fonctions `scanf`, `gets` ou `fgets`, ayant chacune des inconvénients :

- **la fonction `scanf`** pour saisir une chaîne. Ex :

```
scanf ( "%s", chaine );
```

La fonction `scanf` a l'inconvénient majeur de considérer qu'un caractère « espace » est un séparateur. Par exemple, si on tape au clavier « *bonjour tout le monde* », la variable `chaine` ne contiendra que « *bonjour* ».

De plus, la fonction `scanf` n'« avale » pas le « retour à la ligne » (`'\n'`) de fin de saisie (dû à la touche « *Entrée* »). Ce caractère `'\n'` reste présent dans le tampon (*buffer*) d'entrée. Si vous avez une nouvelle instruction `scanf` qui suit la première, elle va lire cet `'\n'` avant toute frappe au clavier, ce qui produit souvent des effets indésirables !

- **la fonction `gets`**. Ex :

```
gets ( chaine );
```

La fonction `gets` est faite spécifiquement pour lire une chaîne de caractères (qui peut contenir des espaces), et elle « avale » le caractère `'\n'` de fin de saisie (sans l'écrire dans la variable `chaine`).

Elle a l'inconvénient de ne pas être sûre : elle ne peut pas empêcher un utilisateur de taper plus de caractères que le nombre `LONGUEUR_MAX_CHAINE` que vous avez prévu. C'est une technique de *hacking* bien connue (appelé « débordement de tampon », *buffer overflow*). Vous pouvez quand même utiliser la fonction `gets` si votre programme est exécuté dans un environnement sûr.

- **la fonction `fgets`**. Ex :

```
fgets ( chaine, LONGUEUR_MAX_CHAINE+1, stdin );
```

La fonction `fgets` est, comme `gets`, faite pour lire (dans un fichier ou au clavier = pseudo-fichier `stdin`) une chaîne de caractères qui peut contenir des espaces, et elle « avale » le caractère `'\n'` de fin de saisie. Toutefois, au contraire de `gets`, elle écrit cet `'\n'` dans la variable `chaine`.

Ce qui la rend plus sûre, c'est qu'elle ne lira pas plus que le nombre de caractères (-1) que vous lui avez donné en second argument.

Ce qui la rend moins pratique que `gets` (en plus d'être plus compliquée à écrire) est l'inconvénient du `'\n'` à fin de la variable `chaine`. Si vous voulez l'enlever, il faut écrire une ligne de code supplémentaire :

```
if (chaine[strlen(chaine)-1] == '\n') chaine[strlen(chaine)-1]='\0';
```

**Remarque importante** : notez bien que ces trois fonctions rajoutent automatiquement le caractère `'\0'` à la fin de la chaîne saisie.

L'affichage d'une chaîne de caractères peut se faire avec les fonctions `printf` ou `puts` :

- la fonction **printf** pour afficher une chaîne. Ex :

```
printf ( "%s", chaine );
```

En général, on ajoute une « partie fixe » pour améliorer la lisibilité à l'écran. Ex :

```
printf ( "Ma chaine : %s\n", chaine );
```

- la fonction **puts**. Elle permet d'afficher directement une chaîne, en y ajoutant automatiquement le retour à la ligne ('\n'). Ex :

```
puts ( chaine );
```

### 3. Les fonctions portant sur les chaînes de caractères

La librairie standard C fournit de nombreuses fonctions pour manipuler les chaînes de caractères. Dans chaque fichier C où on appelle ces fonctions, il faut inclure le *header* standard `string.h`:

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

La ou les chaînes de caractères fournies en argument à ces fonctions sont toujours transmises « par adresse » (c'est à dire par pointeur `char *`). Dans le cas où une fonction utilise une chaîne en argument de sortie (ex: `strcpy`, `strcat`,...), ce sera à vous de prévoir la place mémoire nécessaire.

Voici la liste des fonctions les plus utiles, avec leur prototype et une brève description :

- **strlen**: `size_t strlen ( const char *)` : renvoie la longueur de la chaîne (sans compter le '\0'). Remarque : le type `size_t` correspond à un entier non signé.
- **strcpy**: `char *strcpy ( char *dest, const char *source)` : copie la chaîne `source` dans la chaîne `dest` (y compris le '\0'). **Attention !** : c'est à vous de prévoir que la chaîne `dest` soit assez grande pour recevoir la copie de `source` (un défaut à ce niveau est fréquemment à l'origine de *bugs* dans les programmes C).
- **strncpy**: `char *strncpy ( char *dest, const char *source, int lgmax)` : copie au maximum `lgmax` caractères de la chaîne `source` dans la chaîne `dest` (y compris le '\0', si la longueur de `source` est < `lgmax`). Cette fonction est plus sûre que la précédente, car elle protège contre un éventuel débordement mémoire de la chaîne `dest` si vous fixez correctement `lgmax`. **Attention !** : dans le cas de troncature de `source` à `lgmax`, c'est à vous de gérer le '\0' dans `dest` (il n'est pas ajouté automatiquement).
- **strcat**: `char *strcat ( char *dest, const char *source)` : copie la chaîne `source` à la fin de la chaîne `dest` (concaténation) en écrasant le caractère '\0' à la fin de `dest`, puis en ajoutant un nouveau '\0' final. **Attention !** : c'est à vous de prévoir que la chaîne `dest` soit assez grande pour qu'on puisse lui ajouter la copie de `source` (un défaut à ce niveau est fréquemment à l'origine de *bugs* dans les programmes C).
- **strcmp**: `int strcmp ( const char *ch1, const char *ch2)` : comparaison « lexicographique » entre `ch1` et `ch2`. Renvoie : 0 si `ch1` = `ch2`, un entier négatif si `ch1` < `ch2`, un entier positif si `ch1` > `ch2`.
- **strchr**: `char *strchr ( const char *chaine, char c)` : renvoie l'adresse (un pointeur) de la première occurrence du caractère `c` dans la chaîne `chaine`. Si le caractère `c` n'est pas trouvé, la fonction renvoie le pointeur NULL.
- **strstr**: `char *strstr ( const char *ch1, const char *ch2)` : renvoie l'adresse (un pointeur) de la première occurrence de la chaîne `ch2` dans la chaîne `ch1`. Si la chaîne `ch2` n'est pas trouvée, la fonction renvoie le pointeur NULL.

## 4. Affectation d'une chaîne de caractères

Une erreur fréquente chez les débutants en C est d'écrire l'instruction suivante (on suppose que la variable `chaîne` a été déclarée avant, `char chaîne[10]` ; par exemple) :

```
chaîne = "hello"; // Erreur !
```

Cette instruction, qui sera refusée par le compilateur, ne fait pas ce que peut supposer le programmeur débutant : elle ne recopie pas "hello" dans la variable `chaîne`. En fait elle remplace le contenu de la variable `chaîne` (une adresse) par l'adresse mémoire de la chaîne « constante » "hello". Ici, le compilateur refusera, car l'adresse mémoire d'un tableau ne peut pas être modifiée.

Ce qu'il faut écrire, c'est :

```
strcpy ( chaîne, "hello" );
```

## 5. Lecture/écriture d'un caractère dans une chaîne de caractères

Une chaîne de caractères étant un tableau, on peut accéder à un caractère de la chaîne comme à un élément d'un tableau, par son indice entier (qui débute à 0). **Attention !** : c'est à vous de vérifier que l'indice est inférieur à la taille de la chaîne, sinon on lit (ou écrit) un octet à un emplacement mémoire qui n'a rien à voir (encore une source fréquente de *bug*!). Ex :

```
printf ("%c", chaîne[3] ); //affiche le 4eme caractere de chaîne  
chaîne[0] = 'a'; //ecrase le 1er caractere de chaîne en y mettant 'a'
```

## 6. Comparaison de deux chaînes de caractères

Une autre erreur fréquente chez les débutants en C est d'écrire la comparaison de la façon suivante :

```
if ( chaîne1 == chaîne2 ) { // Erreur !  
    puts ( "Les deux chaînes sont identiques" );  
} else {  
    puts ( "Les deux chaînes sont différentes" );  
}
```

Ces lignes de code seront acceptées par le compilateur, mais ne feront pas ce que peut supposer le programmeur débutant : elles ne comparent pas le contenu des variables `chaîne1` et `chaîne2`. En fait elles comparent l'adresse mémoire de la variable `chaîne1` avec l'adresse mémoire de la variable `chaîne2`.

Ce qu'il faut écrire, c'est :

```
if ( strcmp ( chaîne1, chaîne2 ) == 0 ) { // ou if ( !strcmp(..,..) )  
    puts ( "Les deux chaînes sont identiques" );  
} else {  
    puts ( "Les deux chaînes sont différentes" );  
}
```

## 7. Chaîne de caractères en argument d'une fonction

On peut écrire l'entête d'une fonction recevant une chaîne de caractères de trois façons différentes, quelle que soit la façon dont est déclarée la chaîne avant l'appel de la fonction (cf. §1). Ex :

```
void ma_fction ( char chaîne[10] ) { // a éviter  
void ma_fction ( char chaîne[] ) { // plus clair pour le programmeur?  
void ma_fction ( char *chaîne ) { // plus conforme à la réalité
```

*Remarque* : il est inutile d'avoir un second paramètre pour passer la taille de la chaîne, car le caractère '\0' de la fin permettra toujours à l'ordinateur de savoir où s'arrête la chaîne. Ce serait par contre nécessaire pour un tableau d'entiers ou de réels.

Lequel de ces entêtes choisir ? En fait, et c'est une difficulté du langage C, les trois entêtes seront traduits de la même façon par le compilateur : en un pointeur contenant l'adresse du début de chaîne.

Cependant la première écriture (`chaîne[10]`) est à éviter, car la taille indiquée n'est absolument pas vérifiée par le compilateur (on pourrait mettre n'importe quelle taille, même si elle ne correspond pas à la taille de la chaîne passée effectivement en argument).

La seconde écriture (`chaîne[]`) a le mérite de rappeler au programmeur que la fonction doit traiter un tableau, tout en étant conscient qu'à cet endroit le compilateur voit simplement une adresse.

La troisième écriture (`*chaîne`) est conforme à ce que le compilateur voit : une adresse. C'est au programmeur de savoir ce qu'il doit (et peut) faire avec cette adresse. C'est de cette façon que les fonctions standard déclarent une chaîne de caractères dans leur entête.

**Remarque 1** : quel que soit l'entête, on pourra utiliser l'écriture indicée pour accéder à un élément de la chaîne. Ex :

```
chaîne[1] = 'a';
```

**Remarque 2** : la chaîne étant passée « par adresse » à la fonction, toute modification d'un ou plusieurs caractères sera définitive, même une fois sorti de la fonction. Par contre, **attention** ! la chaîne ne devra pas être redimensionnée à une taille supérieure à sa taille initiale (à la déclaration). Ex (code complet) :

```
#include <stdio.h>
#include <string.h>
#define LONGUEUR_MAX_CHAINE 20

// Fonction de modification d'un caractère
void modif ( char *chaîne ) {
    chaîne[1] = 'a';
}

// Fonction de rajout (concaténation) d'une chaîne constante
void concat ( char *chaîne ) {
    strcat ( chaîne, " world" );// il y aurait eu erreur si
                                // LONGUEUR_MAX_CHAINE valait 10
}

// Programme principal
int main() {
    char tab[LONGUEUR_MAX_CHAINE+1];
    strcpy ( tab, "hello" );

    concat ( tab );
    printf("%s\n", tab ); // affichera hello world
    modif ( tab );
    printf("%s\n", tab ); // affichera hallo world

    return 0;
}
```