Chapitre 13

Chaînes de caractères

Au cours de ce chapitre, nous traiterons de ce qui est appelé, en C, « Chaînes de caractères ».
Nous commencerons par étudier les caractères, qui sont l'élément de base des chaînes de caractères, puis nous verrons comment déclarer une chaîne de caractères. Ensuite, nous apprendrons à les manipuler grâce aux fonctions que fournit TIGCC, et, enfin, nous verrons comment nous pourrions les manipuler par pointeurs.

Tout d'abord, présentons, en quelques mots, ce qu'est une chaîne de caractères, pour ceux qui n'auraient pas l'habitude de ce terme technique.
Le terme de "chaîne de caractères" correspond à un type de données composé d'une suite ordonnée de caractères, et utilisé pour représenter du texte, un caractère étant une lettre, un chiffre, ou un symbole, visible ou non, tel, par exemple, les parenthèses, la ponctuation, ou l'espace.
Nous en avons déjà utilisé sans vraiment savoir qu'il s'agissait de ce dont nous allons parler au cours de ce chapitre.
Par exemple, "ceci est une chaîne de caractères".

I:\ Caractères

Ce que l'on appelle "caractère" est le constituant de base de la chaîne de caractères ; il s'agit du terme utilisé en informatique pour désigner n'importe quel symbole pouvant apparaître dans du texte, que ce soit des lettres, des chiffres, ou n'importe quel autre symbole tel la ponctuation par exemple.
Nos TI codent les caractères sur un octet, en utilisant un codage proche de l'ASCII étendu.
Cela signifie que les 8 bits de chaque octet sont utilisé, ce qui permet 256 codes de caractères différents.
Vous trouverez, page 18 de l'annexe B de votre manuel, disponible en PDF ici si vous ne l'avez pas sous la main, la correspondance entre chaque caractère son code.

A: Notation d'un caractère

En C, pour représenter le code d'un caractère (on dit généralement code ASCII, même si ce n'est pas forcément de l'ASCII "pur"), on écrit le symbole correspondant à celui-ci en le plaçant entre guillemets simples.
Par exemple, le caractère 'a' a pour code, d'après la table de caractères dont nous avons parlé plus haut, 97 en décimal, soit 0x61 en hexadecimal.

Les caractères sont mémorisés, en C, sous forme de variables de type char.
Par exemple, pour déclarer la variable c, de type char, et contenant le caractère 'Y', nous utiliserons cette syntaxe :

```
char c = 'Y';
```

En regardant la table de caractères, nous pouvons observer que les lettres de l'alphabet en majuscules sont toutes à la suite les unes des autres, et qu'il en va de même pour les lettres en minuscules.
Ceci va nous faciliter la vie : pour savoir si un caractère est, par exemple, une lettre majuscule, il nous suffira de tester si son code ASCII est compris entre celui de 'A' et celui de 'Z'.
A titre de curiosité, nous pouvons noter que les lettres minuscules ont des codes ASCII supérieurs à ceux de leurs équivalentes minuscules ; d'ailleurs, l'écart entre le code ASCII d'une lettre minuscule et celui de la lettre majuscule correspondante est, si l'on observe la table, toujours égal à 'a'-'A'.
A partir de là, il nous est facile d'écrire une fonction qui convertisse une lettre majuscule en la lettre minuscule correspondante : tout d'abord, nous testons si le caractère passé en paramètre est bien une lettre majuscule, et, si c'est le cas, nous renvoyons le code de la lettre minuscule correspondante. Si le caractère passé en paramètre n'était pas une lettre majuscule, nous renvoyons ce caractère.
Voici le code source correspondant à un programme implémentant et utilisant une telle fonction :

```
#include <tigcclib.h>

char MAJ_to_min(char c)
{
     if(c>='A' && c<='Z')
          return c+('a'-'A');
     else
          return c;
} // Fin maj_to_min

void _main(void)
{
     clrscr();

     printf("'A' => '%c'\n", MAJ_to_min('A'));
     printf("'F' => '%c'\n", MAJ_to_min('F'));

     ngetchx();
}
```

Exemple Complet

La fonction MAJ_to_min aurait aussi pu être écrite de manière plus concise, comme ceci :

```
char MAJ_to_min(char c)
{
     return c>='A' && c<='Z' ? c+('a'-'A') : c;
} // Fin maj_to_min
```

Notons qu'il existe déjà une fonction dans les librairies de TIGCC qui serve à convertir des caractères de majuscule à minuscule ; il s'agit de la fonction tolower. Sa réciproque est la fonction toupper.

B: Caractères Spéciaux

Les Caractères allant de 0x00 à 0x1F sont généralement appelés caractères non imprimables. Historiquement, ils correspondent à des codes de contrôle utilisés pour, par exemple, les machines à écrire ou les imprimantes en mode texte.
Parmi ceux-ci, on peut par exemple citer le caractère 0x0D, nommé "Carriage Return" (Retour chariot), qui correspond, sur certains systèmes, à un retour à la ligne.
La plupart de ces caractères, sur des machines tels nos ordinateurs, ne sont normalement pas visibles à l'écran, ce qui explique l'appellation de "caractères non imprimables".

Cela dit, la plupart d'entre-eux sont, sur TI, du fait qu'il n'y a pas de périphérique en mode texte à contrôler, représentés par un pictogramme visible à l'écran.

Les caractères de contrôle les plus fréquent peuvent être représentés de manière simple en C : il leur est fait correspondre, au niveau du compilateur, une lettre, précédée d'un antislash pour signifier qu'elle ne doit pas être interprétée telle qu'elle.
En particulier, nous pouvons citer ces quelques caractères :

Caractère	Signification
`'\t'`	Tabulation
`'\n'`	Retour à la ligne
`'\\'`	Antislash
`'\"'`	Guillemets doubles
`'\''`	Guillemet simple

Il en existe d'autres, qui ne sont pas vraiment utile sur TI, puisque nous ne travaillons pas sur un matériel en mode texte ; nous ne les présenterons pas ici.

Nous pouvons remarquer plus particulièrement les trois derniers caractères de notre tableau : ils doivent être précédés d'un antislash, afin que leur signification première soit annulée. Par exemple, le caractère '\", sans l'antislash, correspondrait à la fin du caractère ; on aurait, autrement dit, ceci '", soit deux guillemets n'encadrant aucun caractère, et un guillemet tout seul ne servant à rien... Cela entraînerait une erreur de compilation.
En somme, l'antislash permet de donner une signification différente au caractère qui le suit, ou, comme nous le verrons un peu plus bas, aux caractères qui le suivent. On dit généralement qu'il agit comme caractère d'échappement.
Le fait "d'échapper" un caractère revient donc à le faire précéder d'un antislash.

C: Désigner des caractères par leur code

Il est des caractères qu'il est particulièrement difficile de saisir au clavier. Par exemple, si vous voulez utiliser le caractère '©', vous pouvez avoir quelques difficultés à le trouver.
C'est pour cela que le C permet de désigner des caractères directement par leur code, en octal en préfixant le code par un antislash, ou en hexadécimal en précédant le code par un antislash et un x.
Par exemple, le caractère '©' correspond, d'après la table des codes ASCII trouvée dans le manuel de votre calculatrice, à la valeur 169. 169 en décimal correspond à A9 en hexadécimal, et à 251 en octal.
Au lieu d'écrire '©' dans votre programme, il vous est possible d'utiliser '\251', ou '\xA9', selon la base numérique que vous préférez.
Notez que cela est indispensable pour les caractères non imprimables qui n'ont pas de code textuel. Par exemple, pour afficher un petit verrou, vous utiliserez ceci :

```
#include <tigcclib.h>

void _main(void)
{
	clrscr();

	printf("\16\n");
	printf("\x0E\n");

	ngetchx();
}
```

Exemple Complet
(Du moins, vous utiliserez une des deux écritures, selon votre préférence)

II:\ Chaînes de caractères en C

Maintenant que nous avons vu ce que sont les caractères, nous allons pouvoir commencer à étudier les chaînes de caractères, généralement nommées "string" en anglais.
La norme C défini une chaîne de caractères comme un tableau de chars, dont le dernier élément vaut 0. C'est ce marqueur de fin qui est caractéristique des chaînes de caractères, et toutes les fonctions permettant d'en manipuler supposent que ce 0 final est présent.
Autrement dit, lorsque vous avez une chaîne de caractères contenant 10 lettres, elle occupe en fait 11 emplacements mémoires ; pensez-y lorsque vous avez à déclarer de l'espace mémoire pour stocker une chaîne de caractères.

A: Création de chaînes de caractères

Une chaîne de caractères étant un liste de caractères, elle peut se déclarer comme telle. Par exemple :

```
char str[] = {'S', 'a', 'l', 'u', 't', '\0'};
```

Cela dit, il est extrêmement fastidieux d'utiliser ce type de syntaxe. C'est pourquoi le C fourni les guillemets doubles, qui indiquent au compilateur que ce qu'ils entourent est une chaîne de caractères.
On pourrait ré-écrire notre exemple précédent de cette manière :

```
char str[] = "Salut";
```

Vous remarquerez que, avec cette syntaxe, le 0 de fin de chaîne est automatiquement généré par le compilateur.
Cette écriture est équivalente à celle que nous avons employé juste au-dessus.

Il est aussi possible de déclarer une chaîne de caractères en utilisant la syntaxe par pointeurs, comme ceci :

```
char *str = "Salut";
```

Tant que l'on n'essaye d'accéder à la chaîne de caractères qu'en lecture, ce type de déclaration aura les mêmes conséquences que les deux précédents.
Cela dit, il existe une différence importante entre la déclaration par tableau, et celle par pointeur.

En effet, dans le premier cas (déclaration par tableau), str est un tableau de taille juste suffisamment grande pour pouvoir contenir la chaîne de caractères et son 0 de fin de chaîne. Quelque soit les manipulations effectuées sur la chaîne, str désignera toujours la même zone mémoire. De plus, à chaque fois que vous entrerez dans le bloc au sein de laquelle str est déclarée, la valeur de celle-ci sera réinitialisée.
En revanche, dans le cas de la déclaration par pointeur, le compilateur va créer une zone dans laquelle le texte sera stocké, et initialisera un pointeur pointant sur cette zone. A chaque fois qu'on rentrera dans le bloc contenant cette déclaration, ce sera le pointeur qui sera réinitialisé, et non le texte !

Pour illustrer mes propos, observons deux exemples, dans lesquels nous déclarons, au sein d'une boucle se répétant deux fois, une chaîne de caractères, l'affichons, et modifions son premier caractère.
Dans le premier cas, nous déclarons notre chaîne de caractères par tableau :

```
#include <tigcclib.h>

void _main(void)
{
    short i;

    clrscr();
    for(i=0 ; i<2 ; i++)
    {
        char str[] = "Salut";
        printf("%s\n", str);
        str[0] = '!';
    }
    ngetchx();
}
```

Exemple Complet

Et dans le second cas, nous la déclarons par pointeur :

```
#include <tigcclib.h>

void _main(void)
{
    short i;

    clrscr();
    for(i=0 ; i<2 ; i++)
    {
        char *str = "Salut";
        printf("%s\n", str);
        str[0] = '!';
    }
    ngetchx();
}
```

Exemple Complet

L'écriture str[0] permet d'accéder au premier caractère de la chaîne, puisque, après tout, une chaîne de caractères n'est rien de plus qu'un tableau, quelque soit la façon dont elle est déclarée : ce que nous avons concernant les tableaux au chapitre précédent continue de s'appliquer ici.

Nous obtenons, à l'exécution, les affichages suivants :
Dans le premier cas :

```
Salut
Salut
```

Et dans le second :

```
Salut
!alut
```

Autrement dit, dans le premier cas, la chaîne de caractères a été initialisée à chaque passage dans la boucle, alors qu'au second, elle ne l'a été qu'une et une seule fois ; par la suite, c'est le pointeur, et non la chaîne elle-même, qui a été initialisée.
Ces deux exemples illustrent bien la différence qu'il existe entre les deux écritures.

Cela dit, veuillez noter qu'il **n'est** généralement **pas** une bonne idée que de modifier une chaîne de caractères crée en la plaçant entre guillemets doubles.
En effet, il arrive que le compilateur choisisse, si deux chaînes sont identiques, de les fusionner en une seule. Dans ce cas vous modifiez l'une, cela modifiera aussi l'autre, puisqu'elles ne font finalement qu'une !
Par exemple, regardez cet exemple :

```
#include <tigcclib.h>

void _main(void)
{
    char *str1 = "salut";
    char *str2 = "salut";

    clrscr();

    printf("%s %s\n", str1, str2);

    str1[0] = '!';

    printf("%s %s\n", str1, str2);

    ngetchx();
}
```

Exemple Complet
En premier, on affiche

```
salut salut
```

Mais une fois qu'on modifie str1, on affiche

```
!alut !alut
```

En fait, str2 a aussi été modifiée !
Ce qui nous prouve que le compilateur avait remarqué que str1 et str2 étaient déclarées comme pointant sur le même texte, et avait choisi de ne stocker celui-ci qu'une seule fois dans le programme.
Notez que ce comportement peut varier selon le compilateur. En effet, la norme dit que le comportement en cas de modification d'une chaîne déclarée par pointeur est indéterminé.

Pour en finir avec la déclaration de chaînes de caractères, j'en arrive à la forme qu'on utilise le plus fréquemment lorsque l'on souhaite en manipuler.
Il s'agit d'une déclaration de tableau, dont on précise la taille, mais sans l'initialiser. Encore une fois, il faut penser au 0 de fin de chaîne.
Par exemple, si on veut déclarer une chaîne de caractères qui puisse contenir 10 lettres, on déclarera un tableau de 11 chars, de la manière suivante :

```
char str[11];
```

Maintenant que nous savons comment déclarer des chaînes de caractères, nous allons apprendre, grâce à la suite de ce chapitre, à les manipuler.

B: Manipulations de chaînes de caractères

Les chaînes de caractères étant très souvent employées, et leur format étant strictement défini par la norme C, de nombreuses fonctions permettant de les manipuler sont généralement inclues à aux bibliothèques des compilateurs.
Ainsi, TIGCC et AMS nous fournissent de nombreuses fonctions, dont un bon nombre sont des ROM_CALLs, dédiées au travail avec des chaînes de caractères. Notez que la plupart d'entre elles sont des fonctions ANSI, ce qui garantit que vous les retrouverez avec quasiment tous les compilateurs C existants.
Sous TIGCC, les fonctions de manipulation de chaînes de caractères sont regroupées dans le fichier d'en-tête <string.h>.
Au cours de cette partie, nous allons voir comment copier des chaînes de caractères vers d'autres chaînes de caractères, comment en concaténer, comment en formater, ou encore comment en obtenir la longueur...

1: Longueur d'une chaîne de caractères

Tout d'abord, la fonction strlen permet de déterminer la longueur, en nombre de caractères, de la chaîne de caractères qu'on lui passe en argument. Voici son prototype :

```
unsigned long strlen(const char *str);
```

Notez que c'est le nombre de caractères "utiles" qui est retourné : le 0 de fin n'est pas compté.
Par exemple, considérons cette portion de code :

```
printf("%lu", strlen("salut"));
```

La valeur affichée sera 5 : la chaîne de caractères "salut" contient 5 caractères, même si elle occupe un sixième espace mémoire pour le 0 de fin de chaîne.

2: Copie de chaînes de caractères

Pour copier une chaîne de caractères vers une autre, il convient d'utiliser la fonction strcpy, qui prend en paramètre un pointeur sur la zone mémoire de destination, et la chaîne d'origine :

```
char *strcpy(char *destination, const char *origine);
```

Par exemple :

```
char str[30];
strcpy(str, "Hello World !");
printf("%s", str);
```

Cette portion de code affichera "Hello World !".
Notez que la zone de mémoire de destination doit être de taille suffisamment important pour pouvoir contenir la chaîne de caractères que l'on essaye d'y placer, 0 de fin compris. Dans le cas contraire, votre programme risque de corrompre des données, et d'entraîner un plantage.
Cette fonction retourne un pointeur sur la chaîne de destination.

3: Concaténation de chaînes de caractères

A présent, passons à la concaténation de chaînes de caractères.
En informatique, on utilise le terme "concaténer", pour signifier que l'on veut mettre bout à bout deux chaînes de caractères. Par exemple, la concaténation de "Hello " et de "World !" donnera "Hello World !".
C'est la fonction strcat qui remplit ce rôle. Elle accole à la chaîne de caractères passée en second paramètre à la fin de la première.

```
char *strcat(char *destination, const char *origine);
```

Par exemple :

```
char str[30];
strcpy(str, "Hello");
printf("%s", str);
strcat(str, " World !");
printf("%s", str);
```

Cette portion de code commencera par afficher "Hello", et finira en affichant "Hello World !".
Ici encore, zone de mémoire correspondant à la chaîne de destination doit être de taille suffisamment importante pour pouvoir contenir les textes des deux chaînes de départ, plus le 0 final.

La fonction strcat retourne un pointeur sur la chaîne de destination ; celle qui contiendra le résultat de la concaténation.

Notez que les deux paramètres doivent être des chaînes de caractères valides ! En particulier, le code suivant, où la chaîne de caractères de destination n'a pas été initialisée, ne fonctionnera pas, et risque de causer un plantage :

```
    char str[30];
strcat(str, "Hello World !");
```

En effet, strcat parcourt la première chaîne jusqu'à trouver son 0 final, puis parcourt la seconde chaîne jusqu'à son 0 final, en ajoutant les caractères de celle-ci à la suite de ceux de la première. Donc, si une des deux chaînes n'a pas de 0 final (autrement dit, si un des paramètres n'est pas une chaîne de caractères !), cela ne fonctionnera pas.

4: Comparaison de chaînes de caractères

Assez souvent, lorsque l'on travaille avec des chaînes de caractères, il nous arrive d'avoir à les comparer, pour savoir laquelle est la plus "grande", la plus "petite", ou si elles sont "égales".
Dans ce but, le C fournit la fonction strcmp, qui prend en paramètres deux chaînes de caractères, et retourne :

- 0 si les deux chaînes sont égales.
- Une valeur supérieure à 0 si la première chaîne est plus "grande" que la seconde.
- Une valeur inférieure à 0 si la première chaîne est plus "petite" que la première.

La conversion se faisant sur les codes ASCII de chaque caractères, l'ordre est alphabétique, les majuscules étant plus "petites" que les minuscules.
Voici le prototype de cette fonction :

```
short strcmp(const unsigned char *str1, const unsigned char *str2);
```

Par exemple, pour tester si deux chaînes sont égales, nous pourrons utiliser une écriture telle que celle-ci :

```
if(!strcmp("salut", "salut"))
{
     // Les deux chaînes sont égales
}
```

Je me permet d'insister sur le fait que la valeur retournée en cas d'égalité entre les deux chaînes de caractères est 0.

Pour la forme, voici un petit exemple ou nous testons les 3 cas : égalité, et plus "petit" ou plus "grand" :

```
char str1[] = "aaaab";
char str2[] = "aahbag"
short comp = strcmp(str1, str2);
if(!comp)
    printf("str1 == str2");
else if(comp < 0)
    printf("str1 < str2");
else
    printf("str1 > str2");
```

Si vous êtes curieux, sachez que, en cas de différence entre les deux chaînes, la valeur retournée est égale à la différence entre les codes ASCII du premier caractère différent de chaque chaîne.
Par exemple, si on appelle strcmp sur les chaînes "aaax" et "aazb", la valeur retournée sera 'a'-'z', c'est-à-dire -25 : la chaîne "aaax" est plus petite que la chaîne "aazb".

Si vous avez besoin d'effectuer une comparaison qui ne soit pas sensible à la casse (c'est-à-dire, qui considère que les majuscules et minuscules sont égales), vous pouvez utiliser la fonction suivante :

```
short cmpstri(const unsigned char *str1, const unsigned char *str2);
```

Elle fonctionne exactement comme strcmp, sauf qu'elle est un peu plus lente, étant donné qu'elle convertit tous les caractères de majuscule vers minuscule avant d'effectuer la comparaison.

5: Fonctions travaillant uniquement sur les n premiers caractères

La librairie C fourni aussi des fonctions permettant de copier, concaténer, ou comparer des chaînes, mais en se limitant à un certain nombre de caractères, que l'utilisateur peut définir, en comptant à partir du premier de la chaîne.
Ces fonctions sont nommées comme leurs équivalents travaillant sur des chaînes entières, mais en intercalant un 'n' entre "str" et ce que fait la fonction. Il faut aussi leur passer un troisième paramètre, qui correspond au nombre de caractères que la fonction devra traiter.
Voici leurs prototypes :
Pour la copie des n premiers caractères d'une chaîne vers un bloc mémoire :

```
char *strncpy(char *destination, const char *origine, unsigned long n);
```

Pour accoler à la fin d'une chaîne les n premiers caractères d'une autre :

```
char *strncat(char *destination, const char *origine, unsigned long n);
```

Et enfin, pour comparer les n premiers caractères de deux chaînes :

```
short strncmp(const unsigned char *str1, const unsigned char *str2, unsigned long n);
```

Tout comme celles que nous avons vu précédemment, ces fonctions supposent que les chaînes qu'elles doivent lire sont valides (terminées par un 0), et que les zones mémoires où elles doivent écrire sont suffisamment grandes pour contenir ce que l'on essaye d'y placer.
De plus, comme on peut s'y attendre, si le nombre de caractères que vous spécifiez est plus grand que la taille effective de la chaîne, ces fonctions ignoreront le nombre de vous avez spécifié, et s'arrêteront à la fin de la chaîne.
Notez, et c'est important, que, pour strncpy, si le nombre de caractères dans la chaîne d'origine est supérieur ou égal à n, la chaîne de destination **ne sera pas** terminée par un '\0' ! Dans ce cas, pour que la chaîne de destination soit considérée comme valide, ce qui est indispensable si vous voulez appeler une autre fonction dessus, il faudra que vous rajoutiez le '\0' de fin de chaîne vous-même !

6: Conversion d'une chaîne de caractères vers un entier ou un flottant

Parfois, on dispose d'une chaîne de caractères, et on sait qu'elle contient une écriture de type numérique. On peut alors être amené à vouloir obtenir ce nombre sous forme d'une variable sur laquelle on puisse faire des calculs, telle, par exemple, un entier ou un flottant.
Il existe donc des fonctions qui permettent de convertir des chaînes de caractères en un type numérique. Les voici :
Pour convertir une chaîne de caractères en entier short :

```
short atoi(const char *str);
```

Pour convertir une chaîne de caractères en entier long :

```
long atol(const char *str);
```

Pour ces deux fonctions, on passe en paramètre la chaîne de caractères, et on obtient l'entier en retour. L'analyse de la chaîne de caractères s'arrête au premier caractère non valide pour un entier.
Notez que ces fonctions retourneront un résultat indéterminé si vous essayez de les faire travailler sur une valeur plus grande que ce qu'un short (respectivement, un long) peut contenir.
Si vous avez besoin de plus de flexibilité que celle proposée par ces fonctions, je vous conseille de jeter un coup d'oeil sur la documentation de la fonction strtol. Etant donné qu'il est rare que les débutants aient besoin de ce qu'elle permet de faire, je ne la présenterai pas plus ici.

Et, pour convertir une chaîne de caractères en flottant, on utilisera cette fonction :

```
float atof(const char *str);
```

Elle s'utilise comme les deux fonctions retournant des entiers.
Si la conversion ne peut se faire, atof renverra la valeur NAN (Not A Number). Vous pourrez vérifier si l'appel a échoué en appelant la fonction is_nan sur la valeur retournée par atof.

7: Formatage de chaînes de caractères

Au cours de ce tutorial, nous avons souvent utilisé la fonction printf, qui permet d'afficher à l'écran du texte, en le formatant : insertion du contenu de variables au cours du texte, nombre de caractères fixes, ...
Le C fourni la fonction sprintf, qui fonctionne exactement de la même façon que printf, à la différence prêt que la chaîne de caractères résultante, au lieu d'être affichée à l'écran, est mémorisée dans une variable.
Voici le prototype de cette fonction :

```
short sprintf(char *buffer, const char *format, ...);
```

Le premier paramètre, buffer, correspond à un pointeur vers un tableau de caractères suffisamment grand pour que la fonction puisse y mémoriser la chaîne de caractères formatée.
Le second paramètre, format, correspond à une chaîne de caractères, au sein de laquelle sont utilisés le même type de codes de formatage que pour la fonction printf. Je vous invite à consulter la documentation de printf pour la liste de ces codes.
Ensuite, viennent, dans l'ordre, les paramètres correspondant aux codes de formatage utilisés dans le second paramètre.
Par exemple :

```
char str[30];
short a = 10;
sprintf(str, "a vaut %d\n", a);
```

Naturellement, on peut ensuite travailler avec la chaîne de caractères obtenue ; par exemple, pour l'afficher avec une fonction autre que printf.

III:\ Un peu d'exercice !

Pour terminer ce chapitre, nous allons profiter du fait que certaines fonctions de manipulation de chaînes de caractères soient très simples pour faire un peu d'exercice concernant les tableaux et les pointeurs : nous allons voir comment il est possible de réécrire certaines des fonctions que nous avons ici appris à utiliser.
Pour certaines des fonctions que je choisirai de vous proposer de ré-implémenter, je donnerai une ou plusieurs solutions possibles. En particulier, du moins au début, je proposerai une solution qui pourrait correspondre à du code écrit par un débutant non habitué aux manipulations de pointeurs, et je fournirai une autre solution, correspondant plus à du code de programmeur C ayant de l'expérience dans ce langage. Naturellement, j'expliquerai les différences entre les différentes écritures, et pourquoi on peut passer de l'une à l'autre.
Je ne pense pas que cette partie soit indispensable à pour ce qui est de l'utilisation de base des chaînes de caractères, mais si, un jour, vous avez besoin d'écrire des fonctions travaillant sur du texte, ce que nous allons pourra probablement vous aider. De plus, travailler un peu avec les pointeurs ne peut que vous aider à mieux comprendre leurs principes, si ce n'est pas encore fait. Je vous conseille donc de ne pas négliger les quelques fonctions que nous allons à présenter développer.

J'insiste sur le fait que je ne réimplémente ici ces fonctions qu'à titre d'exercice, et que c'est chose qu'il est absolument ridicule de faire dans un programme utile ! Si des fonctions sont fournies dans les bibliothèques, ce n'est pas pour que vous les redéveloppiez dans vos programmes !

A: strlen

La première fonction que j'ai choisi de réimplémenter est la plus simple de celles que nous avons étudié au cours de ce chapitre ; il s'agit de strlen.
Pour rappel, strlen prend en paramètre une chaîne de caractères, et retourne le nombre de caractères qu'elle contient, sans compter le 0 de fin de chaîne.

Voici la première implémentation que je vous propose de découvrir :

```
unsigned long my_strlen_1(char *str)
{
    unsigned long len = 0;
    while(str[len] != '\0')
    {
        len++;
    }
    return len;
}
```

Le principe est simple, on déclare une variable qui va servir à mémoriser la longueur de la section de chaîne déjà parcourue, et on l'initialise à 0, puisqu'on va commencer le parcours par le début de la chaîne.
Ensuite, on boucle en incrémentant notre compteur de 1 à chaque fois passage dans la boucle, ce qui nous permet de parcourir la chaîne caractère par caractère, du début vers la fin.
Cette boucle dure jusqu'à ce que le caractère courant soit '\0', c'est-à-dire 0. Ce caractère étant le marqueur de fin de chaîne, on met fin au parcours.
Et enfin, on retourne la valeur du compteur de nombre de caractères, qui correspond au nombre de caractères qui précédent le 0 de fin de chaîne.

C'est l'algorithme que nous continuerons d'utiliser pour strlen. Nous ne changerons que la façon d'écrire, afin d'arriver à du code légèrement plus concis.

Première petite modification : le caractère '\0' valant, comme le nom "0 de fin de chaîne" l'indique, zéro, et le 0 étant considéré comme faux dans les tests, on peut ré-écrire la condition de la boucle pour supprimer le test inutile :

```
unsigned long my_strlen_2(char *str)
{
    unsigned long len = 0;
    while(str[len])
    {
        len++;
    }
    return len;
}
```

Le reste de la fonction ne change pas encore.

A présent, dans la condition de la boucle, plutôt que d'accéder au caractère courant de la chaîne en indiçant un tableau, accédons-y par pointeur : à chaque cycle, on incrémente le pointeur correspondant à la chaîne, de sorte à ce qu'il pointe successivement sur le premier, puis le second, puis le troisième, et ainsi de suite.
Et on accède au caractère courant en déréférençant ce pointeur.

```
unsigned long my_strlen_3(char *str)
{
    unsigned long len = 0;
    while(*str++)
    {
        len++;
    }
    return len;
}
```

On a pour l'instant conservé la variable servant à calculer la longueur de la chaîne de caractères, en continuant à l'incrémenter à chaque itération de la boucle.

Il reste une idée que l'on pourrait mettre en application. En effet, en regardant le code de la fonction que nous venons d'écrire, nous réalisons encore deux incrémentations par boucle : celle du pointeur, et celle de la variable destinée à contenir la longueur de la portion de chaîne déjà parcourue. Pourquoi n'essayerions-nous pas de supprimer une de ces deux incrémentations ?
Après tout, cela est possible : supprimons l'incrémentation de la variable len... et la variable aussi, par la même occasion (si on ne l'incrémente plus, elle ne sert à rien).
Maintenant, il nous faut un moyen de déterminer de combien de cases le pointeur a été avancé... Une solution pour cela est de mémoriser, au début de la fonction, sur quelle case il pointe. Et, à la fin de la fonction, nous calculons la différence entre le pointeur str, qui correspond à l'adresse de fin de chaîne, et le pointeur qui a mémorisé l'adresse de début de celle-ci.
Et cette différence, d'après les règles d'arithmétique de pointeurs, correspond au nombre de cases qui séparent les deux pointeurs, c'est-à-dire, au nombre de caractères que la chaîne comporte.
Etant donné que nous avons choisi d'écrire l'instruction d'incrémentation du pointeur directement dans la condition de la boucle, le corps de celle-ci est désormais vide. Nous le remplaçons donc par un point-virgule, qui signifie "instruction vide", plus significatif qu'une paire d'accolades ne contenant rien.

```
unsigned long my_strlen_4(char *str)
{
    char *debut = str;
    while(*str++);
    return str-debut;
}
```

En utilisant cette méthode par rapport à celle que nous employions avec l'implémentation précédente, nous réalisons une soustraction de plus, certes, mais nous écomisons autant d'incrémentations (d'additions, donc) qu'il y a de caractères dans la chanîe de caractères.
Autrement dit, la méthode que nous employons ici a toutes les chances d'être plus rapide que celle utilisée auparavant.

Pour les autres fonctions que nous allons ré-implémenter, nous utiliserons à nouveau ce que nous avons employé ici. Nous ne ré-expliquerons pas tout en détail, puisque nous venons déjà de le faire.

B: strcpy

A présent, nous allons nous pencher sur la fonction de copie de chaînes, strcpy.
Comme nous l'avons vu plus haut, cette fonction prend en paramètres un pointeur sur un bloc mémoire et une chaîne de caractères, et copie la chaîne de caractères vers le bloc mémoire, qui est supposé suffisamment grand pour pouvoir contenir la copie de la chaîne de caractères. Cette fonction retourne un pointeur sur la chaîne de destination.
Globalement, l'algorithme utilisé est simple : on parcourt la chaîne de caractères d'origine caractère par caractère, et, au fur et à mesure, on copie ces caractères vers le bloc mémoire de destination, sans oublier, bien entendu, le 0 de fin de chaîne.

Notre première implémentation est la suivante :

```
char *my_strcpy_1(char *dest, char *orig)
{
    char *debut = dest;
    unsigned short position = 0;
    while(orig[position] != '\0')
    {
        dest[position] = orig[position];
        position++;
    }
    dest[position] = '\0';
    return debut;
}
```

Tout d'abord, nous déclarons un pointeur que nous faisons pointer sur le bloc mémoire de destination, puisque c'est l'adresse de celui-ci que nous devons retourner. Etant donné que, dans cette implémentation, nous ne modifions pas l'adresse de dest, nous aurions pu ne pas utiliser le pointeur debut, et directement retourner dest. Cela dit, par la suite, nous modifierons dest... alors, après tout, autant prendre de bonnes habitudes de suite.
Ensuite, nous déclarons une variable entière qui va servir à mémoriser la position dans les deux chaînes du caractère sur lequel nous travaillons actuellement. Etant donné qu'on parcourt la chaîne en allant du début vers la fin, cette variable de position est initialisée à 0.
La boucle while de notre fonction a un fonctionnement simple : comme pour strlen, nous parcourons la chaîne de caractères caractère par caractère, et nous nous arrêterons au '\0' de fin de chaîne. Dans le corps de la boucle, nous copions le caractère présent à la position courante dans la chaîne d'origine vers la même position dans la chaîne de destination.
Une fois arrivé à la fin de notre boucle, il nous faut ajouter à la fin de la chaîne de destination de '\0' de fin de chaîne, puisque cela n'a pas été fait dans la boucle (une fois qu'on a atteint le 0 de fin de chaîne dans la chaîne d'origine au niveau de la condition de la boucle, on n'entre pas dans son corps, et on ne fait donc pas l'affectation du 0 de fin de chaîne de la chaîne d'origine vers la chaîne de destination... il convient donc de faire cette affectation après la boucle).
Et enfin, on retourne le pointeur sur la chaîne de destination.

Pour notre seconde implémentation de strcpy, nous allons profiter d'une propriété du C, qui dit qu'une affectation a pour valeur celle qu'elle permet de stocker dans son opérande gauche.
Autrement dit,

```
a = b+c
```

vaut a : la somme de b et c est calculée, mémorisée dans a, et toute l'affectation vaut a.

Nous allons déplacer l'affectation depuis le corps de la boucle vers la condition. De la sorte, l'affectation sera fera maintenant avant de tester si on est en fin de chaîne, le test lui-même se faisant en tirant profit de la propriété que nous venons d'énoncer.

```
char *my_strcpy_2(char *dest, char *orig)
{
    char *debut = dest;
    unsigned short position = 0;
    while((dest[position] = orig[position]) != '\0')
    {
        position++;
    }
    return debut;
}
```

De la sorte, ce n'est qu'après avoir copié un caractère de la chaîne d'origine vers la chaîne de destination qu'on vérifiera si ce caractère correspondait au '\0' de fin de chaîne.
Ainsi, nous n'avons plus à placer le '\0' à la fin de la chaîne de destination après la fin de la boucle, comme nous le faisions dans notre implémentation précédente.
Rien d'autre n'a été modifié pour cette version.

A présent, passons à l'utilisation de pointeurs, plutôt que de tableaux et d'indexes.
Les modifications sont proches de celles que nous avons effectué précédemment pour strlen ; je ne les détaillerai donc pas. L'algorithme reste exactement le même que pour notre implémentation précédente.

```
char *my_strcpy_3(char *dest, char *orig)
{
    char *debut = dest;
    while((*dest++ = *orig++) != '\0');
    return debut;
}
```

La variable de position disparaît ; elle n'est plus nécessaire, puisque nous procédons par incrémentation de pointeurs.

Enfin, exactement de la même manière que pour strlen, le test d'égalité entre le caractère courant (résultat de l'affectation) et '0\' est redondant, puisque '\0' vaut 0, et que 0 est équivalent à faux dans un contexte de condition.

```
char *my_strcpy_4(char *dest, char *orig)
{
    char *debut = dest;
    while((*dest++ = *orig++));
    return debut;
}
```

Aucune autre modification n'a été apportée par rapport à l'implémentation précédente.

Encore une fois, utiliser des pointeurs nous a permis d'utiliser de réduire le nombre de lignes de notre fonction, la rendant plus concise.
Cet exemple nous a aussi permis de réaliser l'importance que de petites "astuces" peut prendre. J'ajouterai que ce sont elles qui permettent souvent de faire la différence entre un programmeur C, et un programmeur C expérimenté : ce n'est généralement que lorsque l'on connaît bien le langage C que l'on est à même de se souvenir de certaines de ses particularités de ce genre, qui, même si elles sont loin d'être indispensables à l'écriture de programmes évolués, permettent parfois de se simplifier quelque peu la vie.
(Il aurait probablement été possible de se passer de ce genre d'astuce en utilisant une boucle do...while au lieu d'une boucle while... Mais je souhaitais attirer votre attention sur le fait que le C est un langage plein de subtilités, créées pour permettre des écritures concises, même si leur maîtrise n'est pas nécessairement indispensable.

C: strcat

A présent, je vais vous proposer une implémentation de strcat, qui permet de concaténer deux chaînes de caractères.
Etant donné que le fonctionnement de cette fonction est proche de celles que nous venons de voir, je ne donnerai qu'une seule écriture, et peu d'explications.
En effet, voici ce que strcat fait :

- Tout d'abord, on parcourt la première chaîne de caractères jusqu'à son 0 de fin.
- Ensuite, on recule d'une case, pour que la première écriture de celles qui suivront écrase le 0 de fin de la première chaîne, puisque l'on veut ajouter le texte de la seconde chaîne à la fin de la première.
- A présent, on parcourt la seconde chaîne de caractères, en la copiant au fur et à mesure à la suite des caractères qui étaient déjà présents dans la première.
- Et enfin, on retourne l'adresse du début de la chaîne de destination, qu'on avait pris soin de sauvegarder au début de la fonction.

En somme, la première étape correspond à ce que l'on a vu dans strlen, et les suivantes, à ce qu'on a fait pour strcpy.

Voici l'implémentation de que je vous propose :

```
char *my_strcat_1(char *dest, char *orig)
{
    char *debut = dest;
    while(*dest++); // On se place sur le 0 de fin de chaine
    dest--; // On revient avant le 0 de fin de chaine
    while((*dest++ = *orig++)); // et on copie de orig vers dest
    return debut;
}
```

Si vous ne comprenez pas cet exemple de code, je vous conseille de relire ce que nous avons étudié pour strlen et strcpy : strcat n'est quasiment rien de plus qu'une combinaison du code écrit pour ces deux autres fonctions.

D: strcmp

La dernière fonction que j'ai choisi de vous proposer pour cette partie est la fonction strcmp, qui permet, comme nous l'avons vu précédemment, de comparer deux chaînes de caractères.
Comme nous l'avons dit, si les deux chaînes sont égales, strcmp renvoie 0. Sinon, elle renvoie la différence entre les deux premiers caractères différents d'une chaîne à l'autre.

Pour les quatre implémentations que je vous propose, nous retournons toujours la même chose, que ce soit en écriture avec des tableaux indicés, ou des pointeurs ; je ne reviendrai donc pas dessus : il s'agit simplement de la soustraction évoquée plus haut.

Pour notre première implémentation, nous allons utiliser, pour parcourir nos deux chaînes, des tableaux indicés. Le parcours de chaînes en lui-même n'est pas une nouveauté, puisque nous en avons déjà effectué pour les fonctions précédentes ; je ne reviendrai donc pas dessus en détails.
Voici le code correspondant à notre première version de strcat:

```
short my_strcmp_1(const unsigned char *str1, const unsigned char *str2)
{
    unsigned short position = 0;
    while(str1[position]!='\0' && str2[position]!='\0' && str1[position]==str2
[position])
    {
        position++;
    }
    return str1[position] - str2[position];
}
```

Ce qui est intéressant ici est la condition de la boucle : celle-ci s'exécute tant que :

- `str1[position]!='\0'` : on n'a pas atteint la fin de la première chaîne de caractères.
- `str2[position]!='\0'` : et tant qu'on n'a pas atteint la fin de la seconde chaîne de caractères.
- `str1[position]==str2[position]` : et tant que les caractères des deux chaînes sont égaux.

Autrement dit, on met fin à la boucle si on atteint la fin d'une des deux chaînes, ou si le caractère à la position courante dans la première chaîne est différent du caractère à la position courante dans la seconde.

Pour notre seconde implémentation, nous allons réfléchir un peu plus au sujet de la condition de notre boucle, en vue de l'optimiser. Tout d'abord, nous supprimons les tests avec '\0', qui sont redondants, puisque '\0' est faux.
Mais ensuite, demandons-nous pourquoi faire ces trois tests ? Est-ce qu'on ne pourrait pas n'en faire que deux ?
Certes, il faut vérifier qu'on n'est à la fin ni de la première chaîne, ni de la seconde... Mais pourquoi tester si str1[position] et str2[position] sont non-nuls, alors que nous testons aussi si str1[position] est égal à str2[position] ?
Nous avons plusieurs cas devant nous, en réfléchissant :

- str1[position] est nul. On quitte la boucle sans rien tester d'autres.
- str1[position] est non nul. On teste l'égalité entre str1[position] et str2[position], et s'ils sont différents, on quitte la boucle.
- Mais dans le cas où str2[position] est nul, soit str1[position] sera nul aussi, auquel cas on aura quitté la boucle à cause du premier test, soit str1[position] ne sera pas nul, auquel cas str1[position] sera différent de str2[position], et on quittera la boucle à cause du second test.
  En somme, il n'est pas utile de tester directement si str2[position] est ou non nul ; cela sera de toute façon fait indirectement.

Voici le code correspondant :

```
short my_strcmp_2(const unsigned char *str1, const unsigned char *str2)
{
    unsigned short position = 0;
    while(str1[position] && str1[position]==str2[position])
    {
        position++;
    }
    return str1[position] - str2[position];
}
```

Certes, supprimer **une** condition, ce n'est pas grand chose... Mais **une** condition **sur trois**, c'est déjà une optimisation conséquente, surtout sur une fonction aussi courte !
Encore une fois, nous voyons que sans trop réfléchir, nous pouvons parfaitement écrire des programmes qui fonctionnent... Mais que réfléchir un peu nous permet de les rendre efficaces.

A présent, passons à un accès aux chaînes par pointeurs plutôt que par tableaux. La seule chose qui change est au niveau des écritures, qui nous permettent de nous débarrasser de la variable de position :

```
short my_strcmp_3(const unsigned char *str1, const unsigned char *str2)
{
    while(*str1 && *str1==*str2)
    {
        str1++;
        str2++;
    }
    return *str1 - *str2;
}
```

Aucune autre modification n'a été apportée pour cette version.

Et enfin, uniquement à titre de curiosité, voici comment, en utilisant une boucle for plutôt qu'une boucle while, nous pourrions écrire cette fonction en deux lignes :

```
short my_strcmp_4(const unsigned char *str1, const unsigned char *str2)
{
    for( ; *str1 && *str1==*str2 ; str1++, str2++);
    return *str1 - *str2;
}
```

Il s'agit en fait exactement du même code source, sauf que nous avons changé le type de boucle, tirant profit du fait que la boucle for comprend une partie de test et une autre "d'incrémentation" au sein même de son instruction de boucle.
Cela dit, l'implémentation précédente, utilisant une boucle while, était tout aussi efficace, et sans le moindre doute plus lisible. J'aurais donc, personnellement, tendance à vous conseiller de la préférer à celle-ci : mieux vaut un code source plus lisible, quitte à ce qu'il fasse quelques lignes de plus !

Nous voici arrivé à la fin de ce chapitre du tutorial. Le suivant va nous permettre d'étudier comme regrouper plusieurs données au sein d'une même variable, en utilisant ce que le langage C appelle "structure".