Etude détaillée du module String de la libc

par Franck Hecht (Mon blog)

Date de publication : 19/07/2007

Cet article a pour but d'étudier en profondeur des fonctions du module String *(gestion des chaînes de caractères et des blocs mémoire)* de la libc standard en montrant un algorithme éventuel pour chacune des fonctions et une implémentation naïve adaptée au Langage C !

I - Introduction
- I-A - Remerciements
- I-B - Pré-requis
- I-C - Préparatifs

II - c_memset
- II-A - Prototype
- II-B - Description et comportement
- II-C - Algorithme
- II-D - Implémentation
- II-E - Tests

III - c_memcpy
- III-A - Prototype
- III-B - Description et comportement
- III-C - Algorithme
- III-D - Implémentation
- III-E - Tests

IV - c_memmove
- IV-A - Prototype
- IV-B - Description et comportement
- IV-C - Algorithme
- IV-D - Implémentation
- IV-E - Tests

V - c_memcmp
- V-A - Prototype
- V-B - Description et comportement
- V-C - Algorithme
- V-D - Implémentation
- V-E - Tests

VI - c_memchr
- VI-A - Prototype
- VI-B - Description et comportement
- VI-C - Algorithme
- VI-D - Implémentation
- VI-E - Tests

VII - c_strlen
- VII-A - Prototype
- VII-B - Description et comportement
- VII-C - Algorithme
- VII-D - Implémentation
- VII-E - Tests

VIII - c_strcpy
- VIII-A - Prototype
- VIII-B - Description et comportement
- VIII-C - Algorithme
- VIII-D - Implémentation
- VIII-E - Tests

IX - c_strncpy
- IX-A - Prototype
- IX-B - Description et comportement
- IX-C - Algorithme
- IX-D - Implémentation
- IX-E - Tests

X - c_strcat
- V-A - Prototype

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

X-B - Description et comportement
X-C - Algorithme
X-D - Implémentation
X-E - Tests
XI - c_strncat
XI-A - Prototype
XI-B - Description et comportement
XI-C - Algorithme
XI-D - Implémentation
XI-E - Tests
XII - c_strchr
XII-A - Prototype
XII-B - Description et comportement
XII-C - Algorithme
XII-D - Implémentation
XII-E - Tests
XIII - c_strrchr
XIII-A - Prototype
XIII-B - Description et comportement
XIII-C - Algorithme
XIII-D - Implémentation
XIII-E - Tests
XIV - c_strcmp
XIV-A - Prototype
XIV-B - Description et comportement
XIV-C - Algorithme
XIV-D - Implémentation
XIV-E - Tests
XV - c_strncmp
XV-A - Prototype
XV-B - Description et comportement
XV-C - Algorithme
XV-D - Implémentation
XV-E - Tests
XVI - c_strpbrk
XVI-A - Prototype
XVI-B - Description et comportement
XVI-C - Algorithme
XVI-D - Implémentation
XVI-E - Tests
XVII - c_strspn
XVII-A - Prototype
XVII-B - Description et comportement
XVII-C - Algorithme
XVII-D - Implémentation
XVII-E - Tests
XVIII - c_strcspn
XVIII-A - Prototype
XVIII-B - Description et comportement
XVIII-C - Algorithme
XVIII-D - Implémentation
XVIII-E - Tests
XIX - c_strstr
XIX-A - Prototype

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

XIX-B - Description et comportement
XIX-C - Algorithme
XIX-D - Implémentation
XIX-E - Tests
XX - c_strtok
XX-A - Prototype
XX-B - Description et comportement
XX-C - Algorithme
XX-D - Implémentation
XX-E - Tests
XXI - Le code source complet de l'article
XXI-A - Fichier: c_stddef.h
XXI-B - Fichier: c_string.h
XXI-C - Fichier: c_string.c

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

I - Introduction

Tous les livres habituels soit ne couvrent pas toutes les fonctions du Langage C, soit ne font pas d'études détaillées. Cet article aura donc pour but unique de vous en apprendre plus sur les fonctions de gestion des chaînes de caractères et des blocs mémoire, soit le module *String* standard de la libc.

Outre l'étude de ces fonctions, nous allons voir également un algorithme naïf pour chacune d'entre elles et l'implémentation en Langage C. Ici ne sera pas écrite une implémentation optimisée, mais une implémentation naïve dans un but didactique uniquement afin de mieux comprendre le fonctionnement et cela permettra également de voir comment programmer sans utiliser la libc standard !

Les fonctions étant nombreuses dans ce module, chaque chapitre correspondra à une fonction dont l'organisation sera la suivante:

- Prototype
- Description et Comportement
- Algorithme
- Implémentation
- Tests

Les conventions de nommage utilisées pour nos fichiers, fonctions et autres types sont simples : chaque nom de fonction, type et fichier sera précédé par **c_** ! Il est fait ainsi pour éviter d'utiliser implicitement les fonctions standard au lieu de nos propres fonctions.

I-A - Remerciements

Un grand merci à **hiko-seijuro** pour avoir écrit la partie sur la complexité des algorithmes ainsi qu'à **Alp** et à **julp** pour la relecture attentive et la correction de cet article !

I-B - Pré-requis

Pour suivre cet article confortablement, il est nécessaire d'avoir de très bonnes notions en C, particulièrement sur les pointeurs et l'arithmétique des pointeurs sans oublier des notions sur les chaînes de caractères. Si vous n'avez pas ces connaissances, je vous suggère d'en acquérir les bases en lisant au moins les articles suivants:

- **Les pointeurs du C et du C++** - *par CGi*
- **Les chaînes de caractères en C** - *par Nicolas Joseph*

I-C - Préparatifs

Avant de pouvoir entrer dans le vif du sujet, il nous faut préciser certaines choses. En effet, nous n'allons ici pas utiliser la bibliothèque standard du C, sauf pour le *main* qui nous permettra de tester nos fonctions. Il nous faudra donc un minimum et pour cela, nous allons créer notre propre fichier *stddef.h* donc dans notre cas, le fichier se nommera ***c_stddef.h*** et nous y mettons ce dont nous avons besoin, soit:

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Fichier c_stddef.h

```
#ifndef _H_CSTDDEF
#define _H_CSTDDEF


#undef NULL
#define NULL ((void *) 0)


typedef unsigned int c_size_t;


#endif /* _H_CSTDDEF */
```

Ici nous commençons par redéfinir **NULL** pour mettre en place notre propre implémentation. Ici rien de standard car cette définition peut être sensiblement différente suivant les systèmes.

Nous allons également souvent travailler avec des données de longueurs de chaînes de caractères (entre autres), nous aurons donc aussi besoin du type *size_t* ou dans notre cas **c_size_t** qui est généralement utiliser pour des données de longueurs, des index de tableaux, des retours de fonctions comme *strlen* et bien d'autres.

*Il est à noter que même si nous utilisons ici en réalité un type **unsigned int**, l'implémentation réelle peut être différente suivant les systèmes d'exploitation. Il se peut donc très bien que le type size_t que vous utilisez dans la libc standard couvre un autre type comme par exemple **unsigned long** !*

*Dans tout le document, les prototypes des fonctions se trouveront dans un fichier nommé **c_string.h** et les définitions dans un fichier **c_string.c** !*

Les algorithmes présentés dans les différents chapitres sont des algorithmes génériques et n'utilisent par conséquent aucune notion de pointeurs ce qui aura pour résultat des implémentations sensiblement différentes par rapport aux algorithmes !

*Les fonctions strcoll et strxfrm ont été volontairement omises ! En effet, celles-ci sont étroitement liées à la localisation courante et se basent donc sur la variable d'environnement **COLLATE** (pour strcoll essentiellement) et cela conduirait donc à créer d'autres parties de la libc ce qui n'est pas du tout le but de cet article ! Le premier but de cet article est en effet de programmer soi-même les fonctions de gestion des chaînes de caractères et blocs mémoire de la libc en partant de zéro !*

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

II - c_memset

II-A - Prototype

```
void * c_memset (void * s, int c, c_size_t n);
```

II-B - Description et comportement

La fonction *c_memset* copie l'octet *c* (par conversion de type **unsigned char**) sur une longueur *n* à partir de l'adresse *s*.

La fonction retourne l'adresse *s*.

II-C - Algorithme

Voici un algorithme possible pour la fonction *c_memset*:

```
algorithme
   fonction c_memset (s:générique, c:entier, n:entier):générique
      début
         i <- 0

         tant que n <- n - 1 faire
            s[i] <- c
            i <- i + 1
         ftant

         retourne s
      fin

lexique
   s : générique : Zone mémoire d'un type quelconque.
   c : entier    : Octet à copier.
   n : entier    : Longueur de la copie.
   i : entier    : Variable d'incrémentation.
```

L'algorithme est des plus basiques...

Nous ne disposons que d'une boucle qui part de l'adresse *s*, et avance de *n* cases en mémoire. Les instructions de la boucle copient l'octet *c* à l'emplacement courant de *s* et incrémentent *i* respectivement.

Nous sortons de la fonction en retournant l'adresse *s*.

Complexité temporelle dans le pire des cas:
Parcours simple de la boucle *n* fois: complexité en ***O(n)***

II-D - Implémentation

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
void * c_memset (void * s, int c, c_size_t n)
{
   unsigned char * p_s = s;

   while (n--)
   {
      *p_s++ = c;
   }

   return s;
}
```

L'implémentation change quelque peu par rapport à l'algorithme décrit ci-dessus car nous utilisons des pointeurs (je rappelle que les algorithmes présentés dans cet article sont génériques et peuvent convenir à tout type de langage).

Nous commençons par déclarer un *pointeur sur unsigned char* que nous faisons pointer à l'adresse *s*. La norme spécifie en effet que l'octet *c* (ici de type **int**) est copié par conversion en type **unsigned char** ce qui est fait implicitement ici étant donné le type de notre pointeur *p_s* !

Une chose pratique mais pas trop courante, c'est l'utilisation de l'argument *n* que l'on décrémente directement, cela permet en effet de ne pas avoir à déclarer une variable pour le parcours. Nous décrémentons donc directement la valeur de *n* dans la condition de la boucle.

On termine par copier simplement l'octet *c* à l'emplacement désigné par **p_s* puis nous nous déplaçons dans l'adresse en incrémentant directement notre pointeur !

II-E - Tests

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
   char tab [20] = { 0 };
   int i = 0;

   c_memset (tab, 1, 10);

   for (i = 0; i < 20; i++)
   {
      printf ("%d ", tab[i]);
   }
   printf ("\n");

   return 0;
}
```

Le test est des plus simples : on copie l'octet, ici de valeur **1** sur la première moitié du tableau puis nous l'affichons, voici ce que cela doit donner sur la console :

```
1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0

Process returned 0 (0x0)   execution time : 0.015 s
Press any key to continue.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

III - c_memcpy

III-A - Prototype

```
void * c_memcpy (void * dest, const void * src, c_size_t n);
```

III-B - Description et comportement

La fonction *c_memcpy* copie *n octets* en partant du début de l'adresse *src* vers l'adresse *dest* (en partant également du début de cette adresse).

La fonction a un comportement indéterminé si les deux adresses se chevauchent et ont donc des parties communes. Dans ce cas il est préférable d'utiliser la fonction *c_memmove* !

L'adresse *dest* est renvoyée à la fin de la fonction.

III-C - Algorithme

Voici un algorithme possible pour la fonction *c_memcpy*:

```
algorithme
   fonction c_memcpy (dest:générique, src:générique, n:entier):générique
      début
         i <- 0

         tant que n <- n - 1 faire
            dest[i] <- src[i]
            i <- i + 1
         ftant

         retourne dest
      fin

lexique
   dest : générique : Zone mémoire de destination d'un type quelconque.
   src  : générique : Zone mémoire source d'un type quelconque.
   n    : entier    : Longueur de la copie.
   i    : entier    : Variable d'incrémentation.
```

L'algorithme est simple : nous décrémentons directement la valeur de l'argument *n* (en effet, il n'y a pas de remontée dans la variable d'origine étant donné que sa valeur est copiée, cela peut par contre changer suivant les langages mais c'est le cas en C). La boucle parcourt donc les adresses tant que *n > 0*.

Dans la boucle on copie simplement l'octet courant de l'adresse *src* vers l'adresse *dest* puis nous incrémentons *i* !

On termine par retourner l'adresse *dest* tel qu'il est écrit dans la description de la fonction et donc dans la norme.

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Complexité temporelle dans le pire des cas:
Parcours simple de la boucle *n* fois: complexité en ***O(n)***

III-D - Implémentation

```
void * c_memcpy (void * dest, const void * src, c_size_t n)
{
   char * p_dest = dest;
   const char * p_src = src;

   while (n--)
   {
      *p_dest++ = *p_src++;
   }

   return dest;
}
```

L'implémentation est adaptée au Langage C, les arguments étant de type **void ***, il nous faut les convertir en type **char ***. En effet, le type *char* vaut par définition **1 octet** et ce, quelle que soit l'implémentation du système. Cela convient très bien étant donné que nous voulons copier les adresses octet par octet ! Nous déclarons donc des *pointeurs sur char* pour stocker le début de chaque adresse.

La condition de la boucle ne change pas, nous décrémentons directement l'argument, ce que nous pouvons faire vu qu'il n'y a pas de remontée au-delà de l'argument, cela évite une déclaration de variable supplémentaire.

Le bloc de la boucle copie tout simplement l'octet courant pointé par **p_src* vers l'emplacement pointé par **p_dest* puis nous incrémentons chacune des adresses vers l'emplacement suivant.

On termine en retournant le début de l'adresse *dest* !

III-E - Tests

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
   const char * str1 = "Bonjour, le monde !";
   char str2 [15] = { 0 };

   c_memcpy (str2, str1, 7);

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

   return 0;
}
```

Dans ce petit programme d'exemple, on copie la sous-chaîne *"Bonjour"* de la chaîne *str1* vers le tableau *str2* dont on initialise dès le départ tous les octets à **0** ce qui évitera des désagréments ! En effet, la fonction ne place pas de zéro de fin sauf s'il se trouve dans la longueur *n* donc si votre chaîne de destination n'en possède pas vous aurez des bugs dans l'affichage et même sûrement un comportement indéterminé !

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Voici la sortie sur la console:

```
str1 : Bonjour, le monde !
str2 : Bonjour

Process returned 0 (0x0)   execution time : 0.015 s
Press any key to continue.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

IV - c_memmove

IV-A - Prototype

```
void * c_memmove (void * dest, const void * src, c_size_t n);
```

IV-B - Description et comportement

Le nom de cette fonction peut porter à confusion car en effet, elle ne déplace pas des données mais copie tout comme le fait *c_memcpy* à quelques détails près !

La fonction *c_memmove* est à double usage ! Son comportement est d'une part identique à celui de la fonction *c_memcpy* si les blocs mémoire sont disjoints donc qui ne se chevauchent pas. Là où les deux fonctions diffèrent l'une de l'autre, c'est que la fonction *c_memmove* permet également de prendre en charge des blocs qui se recouvrent partiellement donc qui se chevauchent !

La fonction *c_memmove* copie donc *n octets* de l'adresse *src* vers l'adresse *dest* et retourne l'adresse *dest* !

IV-C - Algorithme

Voici un algorithme possible pour la fonction *c_memmove*:

```
algorithme
   fonction c_memmove (dest:générique, src:générique, n:entier):générique
      début
         si src inférieur ou égal à dest alors
            t1 <- src + (n - 1)
            t2 <- dest + (n - 1)

            tant que n <- n - 1 faire
               dest[t2] <- src[t1]

               t1 <- t1 - 1
               t2 <- t2 - 1
            ftant
         sinon
            c_memcpy (dest, src, n)
         fsi

         retourne dest
      fin

lexique
   dest : générique : Zone mémoire de destination d'un type quelconque.
   src  : générique : Zone mémoire source d'un type quelconque.
   n    : entier    : Longueur de la copie.
   t1   : entier    : Variable de déplacement dans la zone mémoire src.
   t2   : entier    : Variable de déplacement dans la zone mémoire dest.
```

L'algorithme est ici un peu plus long que les autres car il faut gérer deux comportements mais rien d'insurmontable. Dans la condition logique, nous testons le chevauchement des zones mémoire. Si le début de l'adresse *src* est

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

inférieur au début de l'adresse *dest*, il y a un risque de chavauchement des donnéees et donc, pour éviter de corrompre la copie des octets, nous commençons par la fin de la copie de longueur *n* !

Dans cette boucle, nous parcourons un à un les octets de l'adresse *src* en partant de la fin...

*Pour calculer la fin des adresses, on ajoute tout simplement la valeur de l'argument n à l'adresse de la zone à calculer puis on soustrait **1** pour débuter directement à la bonne adresse !*

... puis on les copie dans la zone d'adresse *dest*. A chaque tour de boucle il ne faut pas non plus oublier de décrémenter les variables qui permettent le parcours (oui ici on décrémente car nous partons de la fin jusqu'au début donc à l'envers).

Dans la seconde partie de la condition principale, nous appelons tout simplement la fonction *c_memcpy* car le comportement est alors identique, cela évite de programmer inutilement vu que nous disposons déjà de cette fonction.

On termine la fonction en retournant l'adresse *dest*.

Complexité temporelle dans le pire des cas:
Parcours simple de la boucle <u>n</u> fois: complexité en ***O(n)***

IV-D - Implémentation

```
void * c_memmove (void * dest, const void * src, c_size_t n)
{
   char * p_dest = dest;
   const char * p_src = src;

   if (p_src <= p_dest)
   {
      p_dest += n - 1;
      p_src += n - 1;

      while (n--)
      {
         *p_dest-- = *p_src--;
      }
   }
   else
   {
      c_memcpy (dest, src, n);
   }

   return dest;
}
```

L'implémentation est à peu près identique à l'algorithme sauf qu'elle est adaptée aux pointeurs. Nous ne disposons donc pas de variables de parcours mais nous travaillons directement sur les pointeurs !

IV-E - Tests

```
#include "c_string.h"
```

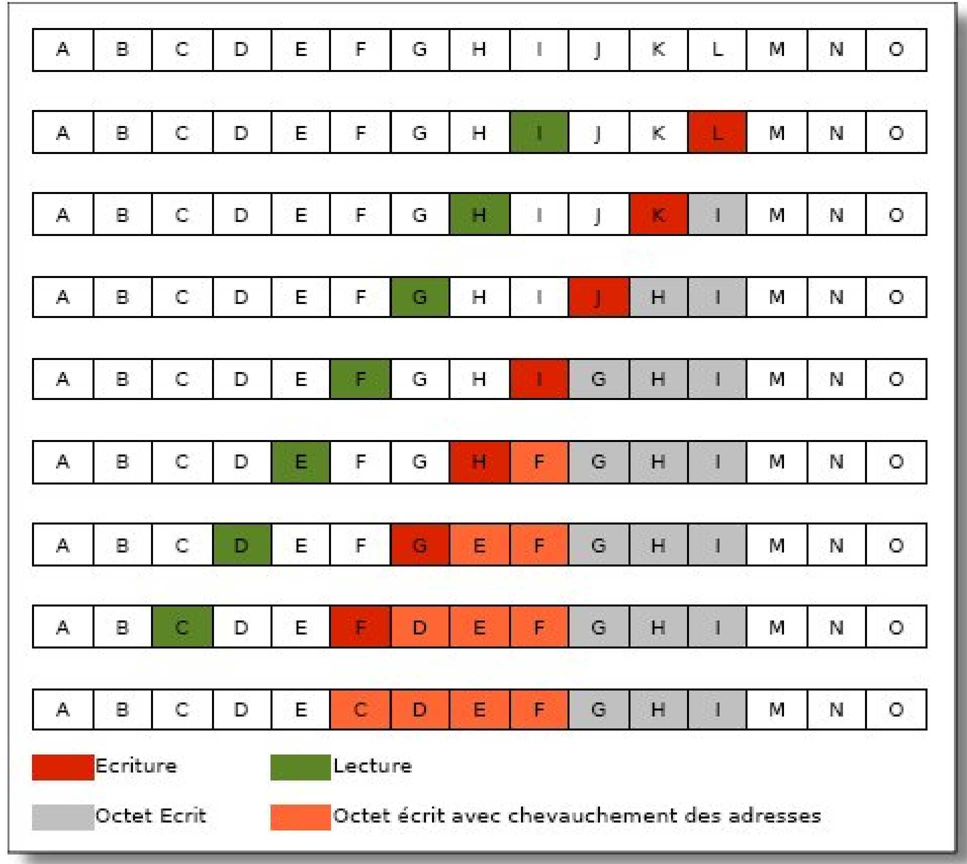

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
#include <stdio.h>


int main (void)
{
   char tab1 [] = "abcdefghijklmno";

   printf ("tab1 = %s\n", tab1);
   c_memmove (tab1 + 5, tab1 + 2, 7);
   printf ("tab1 = %s\n", tab1);

   return 0;
}
```

Dans cet exemple vous pouvez effectivement observer un chevauchement partiel des blocs sur lesquels la fonction doit travailler et dans ce cas, elle va copier les *n octets* en commençant par la fin, voyons le résultat sur la console:

```
tab1 = abcdefghijklmno
tab1 = abcdecdefghimno

Process returned 0 (0x0)   execution time : 0.015 s
Press any key to continue.
```

Nous avons donc fait copier **7** octets en commençant par la fin, ce qui nous fait partir du bloc à l'adresse **tab1 + (2 + 7)** pour commencer à remplir à l'adresse **tab1 + (5 + 7)** et pour arriver en fin d'écriture à l'adresse **tab1 + 5** ! Voici un schéma permettant de mieux visualiser le fonctionnement dans ce cas précis:



Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

V - c_memcmp

V-A - Prototype

```
int c_memcmp (const void * s1, const void * s2, c_size_t n);
```

V-B - Description et comportement

La fonction *c_memcmp* compare chaque octet des deux blocs mémoire *s1* et *s2* sur une longueur *n*. Il faut également noter que le caractère nul participe totalement à la comparaison et par conséquent, une adresse possédant ce caractère sera donc inférieure à l'autre !

La fonction renvoie trois valeurs suivant des cas précis:

- Valeur négative si le bloc d'adresse *s1* est inférieur au bloc d'adresse *s2* dans les *n* premiers octets.
- Zéro si les deux blocs mémoire contiennent les mêmes octets sur les *n* premiers octets.
- Valeur positive si le bloc d'adresse *s1* est supérieur au bloc d'adresse *s2* dans les *n* premiers octets.

La norme du C ne précise cependant pas quelle est la valeur renvoyée. Dans certaines implémentations comme sur Linux et Windows par exemple, cette fonction renvoie les valeurs -1, 0, 1 mais dans d'autres implémentations cette fonction peut directement retourner le résultat de la soustraction des deux octets différents !

*La comparaison des octets se fait par conversion du type **void** * en un type **unsigned char** * !*

V-C - Algorithme

Voici un algorithme possible pour la fonction *c_memcmp*:

```
algorithme
   fonction c_memcmp (s1:générique, s2:générique, n:entier):entier
      début
         i <- 0

         tant que n <- n - 1 faire
            si s1[i] différent de s2[i] alors
               si s1[i] inférieur à s2[i]
                  alors retourne -1
                  sinon retourne 1
               fsi
            fsi

            i <- i + 1
         ftant

         retourne 0
      fin

lexique
   s1 : Zone mémoire quelconque.
   s2 : Zone mémoire quelconque.
   n  : Longueur de la comparaison.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
    i  : Variable d'incrémentation pour le parcours des adresses mémoire.
```

La boucle part du début des adresses *s1* et *s2*, et avance de *n* cases mémoire. Dans la première condition nous testons si les deux octets courants sont différents. Si les deux octets courants sont différents, nous testons leur différence et nous retournons la valeur adéquate.

Nous pouvons voir qu'ici je n'ai pas choisi de retourner comme valeur le résultat d'une soustraction entre les deux octets mais uniquement une valeur relative au bit du signe du résultat comme le font la plupart des implémentations ! Si l'octet pointé par l'adresse *s1[i]* est inférieur à celui de l'adresse *s2[i]* nous retournons la valeur **-1**, **1** sinon.

Si la boucle se termine, c'est la valeur **0** qui est retournée car dans ce cas le contenu des deux adresses est identique !

Complexité temporelle dans le pire des cas:
Parcours simple de la boucle *n* fois: complexité en ***O(n)***

V-D - Implémentation

```
int c_memcmp (const void * s1, const void * s2, c_size_t n)
{
   const unsigned char * p_s1 = s1;
   const unsigned char * p_s2 = s2;

   while (n--)
   {
      if (*p_s1 != *p_s2)
      {
         return *p_s1 < *p_s2 ? -1 : 1;
      }

      p_s1++;
      p_s2++;
   }

   return 0;
}
```

Comme précisé dans la description de la fonction, la comparaison se fait par conversion du type **void *** en type **unsigned char ***, c'est ce que nous faisons dans cette implémentation en faisant pointer les deux adresses par des pointeurs de ce dernier type !

Dans la condition, nous utilisons l'opérateur ternaire en lieu et place d'une simple condition ***if...else*** comme il est décrit dans l'algorithme !

A chaque tour de boucle nous incrémentons également les deux pointeurs *p_s1* et *p_s2* pour avancer sur l'octet suivant !

V-E - Tests

```
#include "c_string.h"
#include <stdio.h>
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
#define TAB_SIZE 4

int main (void)
{
   unsigned char tab1 [TAB_SIZE] = { 'a', 'b', '\0', 'c' };
   unsigned char tab2 [TAB_SIZE] = { 'a', 'b', '\0', 'c' };
   unsigned char tab3 [TAB_SIZE] = { 'a', 'b', 'c', 'h' };
   unsigned char tab4 [TAB_SIZE] = { 'a', 'b', 'a', 'h' };

   printf ("Test 1 : %d\n", c_memcmp (tab1, tab2, TAB_SIZE));
   printf ("Test 2 : %d\n", c_memcmp (tab2, tab3, TAB_SIZE));
   printf ("Test 3 : %d\n", c_memcmp (tab3, tab4, TAB_SIZE));

   return 0;
}
```

Dans ce programme de test nous faisons trois comparaisons pour vérifier tous les points de la fonction *c_memcmp*. Nous commençons par un test d'égalité, un test d'infériorité puis pour finir un test de supériorité ! Voici ce que cela donne en sortie sur la console :

```
Test 1 : 0
Test 2 : -1
Test 3 : 1

Process returned 0 (0x0)   execution time : 0.015 s
Press any key to continue.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

VI - c_memchr

VI-A - Prototype

```
void * c_memchr (const void * s, int c, c_size_t n);
```

VI-B - Description et comportement

La fonction *c_memchr* permet de faire une recherche (par conversion en **unsigned char** *) d'un octet en partant de l'adresse *s*, et de longueur *n*.

Si l'octet est trouvé, la fonction retourne alors l'adresse de ce dernier sinon elle renvoie NULL.

VI-C - Algorithme

Voici un algorithme possible pour la fonction *c_memchr*:

```
algorithme
   fonction c_memchr (s:générique, c:entier, n:entier):générique
      début
         i <- 0

         tant que n <- n - 1 faire
            si s[i] égal à c
               alors retourne s[i]
            fsi

            i <- i + 1
         ftant

         retourne NULL
      fin

lexique
   s : générique : Adresse à partir de laquelle il faut commencer la recherche.
   c : entier    : Octet à retrouver.
   n : entier    : Longueur de la recherche.
   i : entier    : Variable de parcours de l'adresse s.
```

Dans cet algorithme nous parcourons directement l'adresse *s* avec la variable *i* que nous incrémentons à chaque tour de boucle. A chaque tour de boucle on teste également si l'octet courant est identique à l'octet *c*.

Si la condition est vraie alors on retourne l'adresse de l'octet courant. Si l'octet n'est pas trouvé, la boucle se termine et on met fin à la fonction en renvoyant la valeur NULL !

Complexité temporelle dans le pire des cas:
Parcours simple de la boucle *n* fois: complexité en ***O(n)***

VI-D - Implémentation

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
void * c_memchr (const void * s, int c, c_size_t n)
{
   const unsigned char * p_s = s;

   while (n--)
   {
      if (*p_s == c)
      {
         return (void *) p_s;
      }

      p_s++;
   }

   return NULL;
}
```

Dans la description il est précisé que la recherche de l'octet se fait par conversion de l'octet courant de l'adresse *s* en type **unsigned char**, c'est ce que nous faisons en faisant pointer notre adresse *s* par le pointeur *p_s* qui est du type demandé pour la recherche.

Dans le **return** de la condition à l'intérieur de la boucle, il nous faut faire un cast sur la valeur de retour car en effet, on doit renvoyer l'adresse sous la forme d'un type générique donc **void ***.

VI-E - Tests

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
   char * str = "Une chaine de caracteres !";
   char * ret = NULL;

   ret = c_memchr (str, 'd', 12);
   printf ("ret = %s\n", ret);

   return 0;
}
```

Dans ce programme d'exemple, nous recherchons l'octet désigné par **'d'** dans la chaîne de caractères *str* puis nous faisons retourner le résultat vers le pointeur *ret* et affichons le reste de la chaîne à partir du début pointé par *ret* !

Ceci est à peu près le fonctionnement des fonctions *c_strchr* et *c_strrchr* mais celles-ci ne sont destinées qu'aux chaînes de caractères alors que la fonction *c_memchr* peut être appliquée à n'importe quel type de données !

Voici la sortie du programme ci-dessus:

```
ret = de caracteres !

Process returned 0 (0x0)   execution time : 0.015 s
Press any key to continue.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

VII - c_strlen

VII-A - Prototype

```
c_size_t c_strlen (const char * s);
```

VII-B - Description et comportement

La fonction *c_strlen* retourn la longueure de la chaîne de caractères passée en argument. Le *"zéro de fin de chaîne"* n'est pas inclus dans le résultat retourné lors de la fin de la fonction.

En cas d'échec ou de chaîne vide, la fonction renvoie simplement la valeur ***0** (zéro)* !

Si la chaîne de caractères ne contient pas de *zéro de fin*, le comportement de la fonction est indéterminé et peut éventuellement générer un plantage du programme car il y a un risque de dépassement de la zone mémoire de la chaîne passée en argument.

VII-C - Algorithme

Voici un algorithme possible pour la fonction *c_strlen*:

```
algorithme
   fonction c_strlen (s:chaîne):entier
      début
         taille <- 0
         i <- 0

         tant que s[i] différent de '\0' faire
            taille <- taille + 1
            i <- i + 1
         ftant

         retourne taille
      fin

lexique
   s      : chaîne : Chaîne de caractères dont il faut calculer la taille.
   i      : entier : Variable d'incrémentation pour se déplacer dans la chaîne.
   taille : entier : Compteur de la taille de la chaîne.
```

L'algorithme est très simple : dans un premier temps nous déclarons des variables entières pour stocker la taille de la chaîne et la position sur le caractère courant de la chaîne.

La boucle quant à elle permet de parcourir la chaîne caractère par caractère tant que le caractère courant n'est pas égal au caractère de fin de chaîne. A chaque itération de la boucle, les variables *taille* et *i* sont incrémentées de **1**.

Une fois le caractère de fin de chaîne trouvé, la boucle se termine et la fonction renvoie la valeur de la variable *taille* !

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Complexité temporelle dans le pire des cas:
Parcours simple de la boucle *taille* fois: complexité en **O(taille)**

VII-D - Implémentation

Dans notre implémentation, nous allons un peu réduire l'algorithme ci-dessus car certaines opérations peuvent êtres effectuées dans la même instruction grâce à l'arithmétique des pointeurs.

```
c_size_t c_strlen (const char * s)
{
   c_size_t size = 0;

   while (*s++)
   {
      size++;
   }

   return size;
}
```

La condition de la boucle se trouve donc accompagnée d'une instruction d'incrémentation qui incrémente la position du pointeur après avoir lu le caractère courant ce qui nous évite surtout de déclarer une variable supplémentaire comme il est indiqué dans l'algorithme. Nous utilisons pour valeur de retour le type *c_size_t* au lieu d'un simple entier car ce type est prévu à cet effet.

VII-E - Tests

Testons notre fonction avec un programme simple. Nous déclarons une chaîne que nous passons ensuite à notre fonction *c_strlen* qui stocke la taille de la chaîne dans la variable *size*. Pour finir notre court programme, nous affichons cette valeur avec la fonction *printf*:

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
   const char * str  = "Ma chaine de caracteres !";
   c_size_t     size = c_strlen (str);

   printf ("%s = %d char\n", str, size);

   return 0;
}
```

Nous avons donc si tout va bien, le résultat suivant:

```
Ma chaine de caracteres ! = 25 char
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

VIII - c_strcpy

VIII-A - Prototype

```
char * c_strcpy (char * dest, const char * src);
```

VIII-B - Description et comportement

La fonction *c_strcpy* copie les caractères de la chaîne *src* vers l'adresse de la chaîne *dest*, le *zéro de fin de chaîne* est également copié.

Si les deux chaînes se chevauchent ou si la chaîne *src* ne contient pas de *zéro final*, le comportement du programme est indéterminé.

La fonction renvoie l'adresse de la chaine *dest*.

VIII-C - Algorithme

Voici un algorithme possible pour la fonction *c_strcpy*:

```
algorithme
   fonction c_strcpy (dest:chaîne, src:chaîne):chaîne
      debut
         i <- 0

         tant que dest[i] <- src[i] faire
            i <- i + 1
         ftant

         retourne dest
      fin

lexique
   dest : chaîne : Chaine de destination, adresse renvoyée par la fonction.
   src  : chaîne : Chaine source.
   i    : entier : Variable d'incrémentation pour se déplacer dans la chaîne src.
```

Nous commençons par déclarer une variable qui permet de se déplacer à l'intérieur de la chaîne *src* et *dest* caractère par caractère.

La condition d'arrêt de la boucle est si le dernier caractère de la chaîne *src* est copié. La condition de la boucle est également l'instruction de copie du caractère courant. Nous pouvons en effet remarquer que l'algorithme copie dans la condition de la boucle, le caractère courant, y compris le *zéro de fin* !

Le bloc d'instruction de la boucle incrémente ici la variable *i* pour se placer sur le caractère suivant.

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Une fois le dernier caractère copié, ici le caractère de fin, la fonction se termine en retournant l'adresse de la chaîne *dest*.

Complexité temporelle dans le pire des cas:
Parcours simple de la boucle *taille(src)* fois: complexité en **O(taille(src))**

VIII-D - Implémentation

```
char * c_strcpy (char * dest, const char * src)
{
   while ((*dest++ = *src++));
   return dest;
}
```

L'implémentation est dans notre cas, en langage C, un peu plus courte. Ce rétrécissement est dû aux pointeurs. En effet, nous n'utilisons ici aucune variable de déplacement dans les chaînes mais nous incrémentons directement l'adresse des pointeurs et par conséquent, directement ceux des arguments de notre fonction.

Le déroulement en est simple, voici l'ordre:

1. Copie du caractère courant (*dest = *src).
2. Incrémentation de l'adresse de *dest* et *src* (dest++, src++).

La condition d'arrêt de la boucle se pose d'elle même lorsque l'instruction copie le dernier caractère. En effet, la boucle n'ayant plus de caractères à copier, elle se termine d'elle même !

VIII-E - Tests

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
   const char str1 [26] = "Ma chaine de caracteres !";
   char str2 [26];

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

   return 0;
```

Ce simple programme permet de tester la copie d'une chaîne de caractères avec notre fonction *c_strcpy*. Pour tester les cas par exemple où la chaîne source *str1 (str2 étant ici la chaîne de destination de la copie)* n'a pas de *zéro de fin*, mettez simplement la valeur du nombre de caractères du tableau à **25** ce qui permet de voir dans le meilleur des cas que le caractère final est bien pris en compte dans la copie !

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

IX - c_strncpy

IX-A - Prototype

```
char * c_strncpy (char * dest, const char * src, c_size_t n);
```

IX-B - Description et comportement

La fonction *c_strncpy* copie les *n premiers* caractères de la chaîne *src* vers la chaîne *dest*. Le *zéro de fin de chaîne* n'est pas copié contrairement à la fonction *c_strcpy* !

Plusieurs comportements suivant des cas précis sont à noter, les voici:

1. Si la longueur *n* est inférieure à la longueure de la chaîne *src* alors, la chaîne *dest* ne possèdera pas de caractère de fin de chaîne.
2. Si la longueur *n* est égale à la longueure de la chaîne *src* alors, la chaîne *dest* ne possèdera également pas de zéro final.
3. Si la longueur *n* est supérieure à la longueure de la chaîne *src* alors, la chaîne *dest* se verra attribuer des caractères nuls *(zéro de fin)* jusqu'à ce que la longueure *n* soit atteinte.

IX-C - Algorithme

Voici un algorithme possible pour la fonction *c_strncpy*:

```
algorithme
   fonction c_strncpy (dest:chaîne, src:chaîne, n:entier):chaîne
      debut
         i <- 0

         si n supérieur à 0 alors
            pour i de 0 à n faire
               si src[i] différent de '\0'
                  alors dest[i] <- src[i]
                  sinon fpour
               fsi
            fpour

            si i inférieur à n alors
               faire
                  dest[i] <- '\0'
                  i <- i + 1
               tant que i < n
            fsi
         fsi

         retourne dest
      fin

lexique
   dest : chaîne : Chaine de destination, adresse renvoyée par la fonction.
   src  : chaîne : Chaine source.
   n    : entier : Nombre de caractères à recopier.
   i    : entier : Variable d'incrémentation pour se déplacer dans la chaîne src.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Comme à l'accoutumée, nous commençons par déclarer une variable d'incrémentation *(ici "i")* et que nous initialisons à **0** !

Dans cet algorithme, nous mettons la suite complète des instructions restantes dans une seule condition principale ce qui évite de faire plusieurs *return* au sein d'une même fonction et c'est d'autant plus propre.

Dans la première boucle, nous copions simplement les caractères un par un de la chaîne *src* vers *dest* de **0 à n**, *n* étant le nombre de caractères à copier. La copie se poursuit tant que le caractère courant ne correspond pas au caractère *nul* donc au *zéro de fin chaîne* car comme il a été dit plus haut dans la description, ce caractère de la chaîne *src* n'est pas copié !

Dans la seconde boucle, nous remplissons simplement le reste de la chaîne *dest* avec des caractères nuls si bien sûr, nous n'avons pas atteint la longueur *n* dans la première boucle, le remplissage se fait alors jusqu'à atteindre cette longueur !

Pour finir, nous sortons de la fonction en retournant l'adresse de la chaîne *dest*.

Complexité temporelle dans le pire des cas:
Si la première boucle ne va pas jusqu'à *n* alors la seconde s'en charge: complexité en **O(n)**

IX-D - Implémentation

Il n'y a rien de plus à préciser sur cette implémentation car elle reflète exactement l'algorithme présenté ci-dessus !

```
char * c_strncpy (char * dest, const char * src, c_size_t n)
{
   c_size_t i = 0;

   if (n > 0)
   {
      for (i = 0; i < n; i++)
      {
         if (src[i] != '\0')
         {
            dest[i] = src[i];
         }
         else
         {
            break;
         }
      }

      if (i < n)
      {
         do
         {
            dest[i] = '\0';
            i++;
         }
         while (i < n);
      }
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
    }

    return dest;
}
```

IX-E - Tests

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
    const char str1 [26] = "Ma chaine de caracteres !";
    char str2 [30];
    char str3 [30];
    char str4 [30];

    c_strncpy (str2, str1, 20); /* Cas 1 */
    c_strncpy (str3, str1, 25); /* Cas 2 */
    c_strncpy (str4, str1, 30); /* Cas 3 */

    printf ("str1 : %s\n", str1);
    printf ("str2 : %s\n", str2);
    printf ("str3 : %s\n", str3);
    printf ("str4 : %s\n", str4);

    return 0;
}
```

Dans ce programme de test, nous réalisons les tests sur les trois cas cités dans la description de la fonction *c_strncpy* soit:

1. Nous copions *20* caractères pour vérifier si la chaîne *dest* possède un caractère nul lorsque la fonction se termine.
2. Nous copions *25* soit la longueur exacte de la chaîne *dest* sans compter le caractère de fin de chaîne, pour tester le même comportement que le cas 1.
3. Nous copions *30* caractères donc dans ce cas une longueur *n* supérieure à la taille de la chaîne *src* pour vérifier que la chaîne *dest* sera complétée avec des caractères nuls.

Vous devriez vous retrouver avec une sortie de ce genre mais qui peut être sensiblement différente sur chaque ordinateur/système :

```
str1 : Ma chaine de caracteres !
str2 : Ma chaine de caractep ¥w    Ò-+wMa chaine de caracteres !
str3 : Ma chaine de caracteres ! ¥w    Ma chaine de caractep ¥w    Ò-+wMa chaine de caracteres !
str4 : Ma chaine de caracteres !

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

X - c_strcat

V-A - Prototype

```
char * c_strcat (char * dest, const char * src);
```

X-B - Description et comportement

La fonction *c_strcat* permet de concaténer la chaîne *src* à la fin de la chaîne *dest* donc à partir du *zéro de fin* de cette dernière.

Si la chaîne *src* est vide alors la chaîne *dest* ne sera pas changée.

La concaténation commençant à la fin de la chaîne *dest*, si celle-ci ne dispose pas d'un caractère de fin de chaîne, la fonction continue la recherche de ce dernier jusqu'à en trouver un. Ceci a pour conséquence un comportement indéterminé de votre programme car l'écriture de la chaîne *src* se fera alors dans une zone mémoire quelconque et donc écrasement de données en mémoire !

Il faut par conséquent s'assurer que les chaînes de caractères possèdent un zéro terminal et également que la chaîne *dest* soit assez grande pour stocker la chaîne *src* en plus de ses propres caractères et bien sûr le zéro de fin !

X-C - Algorithme

Voici un algorithme possible pour la fonction *c_strcat*:

```
algorithme
   fonction c_strcat (dest:chaîne, src:chaîne):chaîne
      debut
         i <- 0
         j <- longueur (dest)

         faire
            dest[j] <- src[i]

            j <- j + 1
            i <- i + 1
         tant que src[i] différent de '\0'

         retourne dest
      fin

lexique
   dest : chaîne : Chaine de destination, adresse renvoyée par la fonction.
   src  : chaîne : Chaine source.
   i    : entier : Variable d'incrémentation pour se déplacer dans la chaîne src.
   j    : entier : Variable d'incrémentation pour se déplacer dans la chaîne dest.
```

L'algorithme est assez basique... Nous commençons par déclarer deux variables pour le déplacement dans les chaînes *dest* et *src*. Ici **i** pour la chaîne *src* que nous initialisons à **0** et **j** pour la chaîne *dest* que nous initialisons à la longueur de la chaîne *dest*, on se positionne donc sur le caractère de fin de celle-ci.

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Dans la boucle, nous copions simplement caractère par caractère de la chaîne *src* vers la chaîne *dest* puis nous incrémentons nos variables pour passer aux emplacements mémoire suivants.

On termine la fonction en retournant l'adresse de la chaîne *dest*.

Complexité temporelle dans le pire des cas:
On parcourt simplement toute la chaîne à concaténer: complexité en **O(taille(src)+C(c_strlen(dest))**

X-D - Implémentation

```
char * c_strcat (char * dest, const char * src)
{
   const char * p1 = src;
   char * p2 = dest + c_strlen (dest);

   do
   {
      *p2++ = *p1;
   }
   while (*p1++);

   return dest;
}
```

L'implémentation ci-dessus change un petit peu par rapport à l'algorithme que nous venons de voir car nous utilisons des pointeurs et par conséquent, nous n'avons pas besoin de variables d'incrémentation, nous utilisons à la place et comme d'habitude l'arithmétique des pointeurs.

Nous déclarons pour commencer deux pointeurs qui nous servirons à nous déplacer dans les chaînes de caractères. Le premier *p1*, commence sur le premier caractère de la chaîne *src* et le second *p2* commence lui à la fin de la chaîne *dest*.

Vient ensuite la boucle de copie. On peut remarquer que la fonction incrémente le pointeur *p2* dans le corps de la boucle et le pointeur *p1* dans la condition de la boucle. Ceci permet en effet de copier le *zéro de fin* de la chaîne pointée par *p1* avant de sortir de la boucle, cela évite une instruction supplémentaire pour insérer un caractère de fin en dehors de la boucle !

Comme d'habitude, on quitte la fonction en retournant l'adresse de la chaîne *dest*.

X-E - Tests

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
   char str1 [30] = "Ma chaine ";
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
    const char str2 [] = "de caracteres !";

    c_strcat (str1, str2);

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

    return 0;
}
```

Dans ce programme de test, on concatène simplement la chaîne *str2* à la suite de la chaîne *str1* et on affiche le résultat !

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

XI - c_strncat

XI-A - Prototype

```
char * c_strncat (char * dest, const char * src, c_size_t n);
```

XI-B - Description et comportement

La fonction *c_strncat* concatène *n caractères* de la chaîne *src* vers la chaîne *dest*. Si la fonction rencontre un caractère de fin de chaîne durant les *n caractères* elle s'arrête et sinon, la fonction ne copie pas plus que la longueur *n*.

Quel que soit le cas de figure, la fonction ajoute un caractère nul pour marquer la fin de la chaîne *dest* !

Si la chaîne *src* est vide alors la chaîne *dest* sera inchangée.

La concaténation commençant à la fin de la chaîne *dest*, si celle-ci ne dispose pas d'un caractère de fin de chaîne, la fonction continue la recherche de ce dernier jusqu'à en trouver un. Par conséquent l'écriture se fera alors dans une zone de mémoire quelconque pouvant entraîner un comportement indéterminé de votre programme en plus d'un potentiel écrasement des données en mémoire.

Il faut par conséquent s'assurer que les chaînes de caractères possèdent un zéro terminal et également que la chaîne *dest* soit assez grande pour stocker la chaîne *src* en plus de ses propres caractères et bien sûr son zéro de fin !

XI-C - Algorithme

Voici un algorithme possible pour la fonction *c_strncat*:

```
algorithme
   fonction c_strncat (dest:chaîne, src:chaîne, n:entier):chaîne
      debut
         i <- 0
         j <- longueur (dest)

         faire
            dest[j] <- src[i]

            j <- j + 1
            i <- i + 1
         tant que n <- n - 1 et src[i] différent de '\0'

         retourne dest;
      fin

lexique
   dest : chaîne : Chaine de destination, adresse renvoyée par la fonction.
   src  : chaîne : Chaine source.
   n    : entier : Nombre de caractères à copier.
   i    : entier : Variable d'incrémentation pour se déplacer dans la chaîne src.
   j    : entier : Variable d'incrémentation pour se déplacer dans la chaîne dest.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Nous commençons par déclarer deux variables pour le déplacement dans les chaînes *dest* et *src*. Ici **i** pour la chaîne *src* que nous initialisons à **0** et **j** pour la chaîne *dest* que nous initialisons à la longueur de la chaîne *dest*, on se positionne donc sur le caractère de fin de celle-ci.

La boucle parcourt de **0** jusqu'à **n** les caractères de la chaîne *src* ou jusqu'à trouver un caractère de fin de chaîne (en réalité cet algorithme est identique à celui de la fonction *c_strcat* à part l'instruction de décrémentation qui vient s'ajouter dans la condition de la boucle).

Dans le corps de la boucle nous copions le caractère courant et nous incrémentons les variables *i* et *j*.

A la sortie de la boucle nous retournons l'adresse de la chaîne *dest*.

Complexité temporelle dans le pire des cas:
Dans le pire des cas *n>=taille(src)*: complexité en **O(taille(src)+C(c_strlen(dest))**

XI-D - Implémentation

```
char * c_strncat (char * dest, const char * src, c_size_t n)
{
   const char * p1 = src;
   char * p2 = dest + c_strlen (dest);

   do
   {
      *p2++ = *p1++;
   }
   while (n-- && *p1);

   return dest;
}
```

Tout comme la fonction *c_strcat* précédente, l'implémentation change un petit peu car elle est adaptée à l'utilisation des pointeurs, je n'entrerai donc pas dans les détails d'implémentation ici car ceci a déjà été dans le chapitre précédent !

XI-E - Tests

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
   char str1 [30] = "Ma chaine ";
   const char str2 [] = "de caracteres !";

   c_strncat (str1, str2, 14);

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

   return 0;
}
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Dans ce programme de test on ne concatène qu'une partie de la chaîne *str2* donc ici jusqu'au caractère **s**. On peut tester également le comportement de la fonction si elle trouve un *zéro de fin* durant les *n caractères* comme par exemple avec une chaîne *str2* ci-dessous:

```
const char str2 [] = "de carac\0teres !";
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

XII - c_strchr

XII-A - Prototype

```
char * c_strchr (const char * s, int c);
```

XII-B - Description et comportement

La fonction *c_strchr* permet de rechercher la première occurrence du caractère *c* (par conversion du type entier en caractère) dans la chaîne *s* passée en argument. Le zéro de fin participe à la recherche ainsi, lorsque vous passez le caractère nul comme caractère à rechercher, la fonction renvoie un pointeur sur la fin de la chaîne !

Si le caractère passé en argument ne figure pas dans la chaîne, la fonction renvoie la valeur NULL.

La chaîne de caractères doit posséder un caractère de code nul sinon il y a risque de dépassement de la zone mémoire passée en argument ce qui peut induire un comportement indéterminé de votre programme !

XII-C - Algorithme

Voici un algorithme possible pour la fonction *c_strchr*:

```
algorithme
   fonction c_strchr (s:chaîne, c:entier):chaîne
      debut
         i <- 0

         tant que s[i] différent de c faire
            si s[i] == '\0'
               alors retourne NULL
            fsi

            i <- i + 1
         ftant

         retourne s[i]
      fin

lexique
   s : chaîne : Chaîne dans laquelle il faut trouver la première occurrence du caractère c.
   c : entier : Caractère dont il faut trouver la première occurrence dans la chaîne s.
   i : entier : Variable d'incrémentation pour se déplacer dans la chaîne s.
```

L'algorithme est ici très basique, nous parcourons simplement la chaîne *s* jusqu'à trouver le caractère passé en second argument. Si on rencontre le caractère de fin de chaîne avant de trouver la première occurrence du caractère qui est recherché, la fonction se termine en renvoyant la valeur NULL.

A chaque tour de boucle on incrémente *i* et on fait le test pour déterminer si l'on se trouve à la fin de chaîne.

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

On termine par renvoyer l'emplacement mémoire du caractère si une occurrence existe.

Complexité temporelle dans le pire des cas:
Dans le pire des cas où on ne trouve pas le caractère: complexité en **O(taille(s))**

XII-D - Implémentation

```
char * c_strchr (const char * s, int c)
{
   const char * p = s;

   while (*p != (char) c)
   {
      if (*p == 0)
      {
         return NULL;
      }

      p++;
   }

   return (char *) p;
}
```

L'implémentation est ici légèrement différente car adaptée aux pointeurs et donc au C.

Au lieu d'utiliser une variable d'incrémentation nous utilisons un pointeur qui permet de se déplacer dans la chaîne de caractères. Nous pouvons remarquer que le caractère à rechercher est un entier tel qu'il l'a été dit dans la description, il nous faut donc convertir par cast cette valeur afin de pouvoir effectuer le test correctement.

XII-E - Tests

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
   const char * str = "Ma chaine de caracteres !";
   char * p = NULL;

   p = c_strchr (str, 'c');

   if (p != NULL)
   {
      printf ("%s\n", p);
   }

   return 0;
}
```

Le test est simple, nous recherchons le caractère **c** dans la chaîne **str** puis nous affichons le résultat ce qui correspond à la sortie suivante:

```
chaine de caracteres !
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
Process returned 0 (0x0)   execution time : 0.015 s
Press any key to continue.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

XIII - c_strrchr

XIII-A - Prototype

```
char * c_strrchr (const char * s, int c);
```

XIII-B - Description et comportement

La fonction *c_strrchr* permet de rechercher la dernière occurrence du caractère *c* (par conversion du type entier en caractère) dans la chaîne *s* passée en argument. Le zéro de fin participe à la recherche ainsi, lorsque vous passez le caractère nul comme caractère à rechercher, la fonction renvoie un pointeur sur la fin de la chaîne !

Si le caractère passé en argument ne figure pas dans la chaîne, la fonction renvoie la valeur NULL.

La chaîne de caractère doit posséder un caractère de code nul sinon il y'a risque de dépassement de la zone mémoire passée en argument ce qui a pour effet une possibilité d'un comportement indéterminé de votre programme !

XIII-C - Algorithme

Voici un algorithme possible pour la fonction *c_strrchr*:

```
algorithme
   fonction c_strrchr (s:chaîne, c:entier):chaîne
      debut
         i <- longueur (s)

         faire
            si s[i] égal à c
               alors retourne s[i]
            fsi

            i <- i - 1
         tant que i supérieur ou égal à 0

         retourne NULL
      fin

lexique
   s : chaîne : Chaîne dans laquelle il faut trouver la dernière occurrence du caractère c.
   c : entier : Caractère dont il faut trouver la première occurrence dans la chaîne s.
   i : entier : Variable d'incrémentation pour se déplacer dans la chaîne s.
```

Nous avons ici l'algorithme inverse de la fonction *c_strchr*.

On commence par récupérer la taille de la chaîne que nous stockons dans la variable *i*. Dans la boucle, on débute par le test du dernier caractère de la chaîne ; s'il correspond au caractère à rechercher, on quitte la fonction en retournant l'adresse de ce caractère dans la chaîne.

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

A chaque tour de boucle, nous effectuons cette vérification et si la condition n'est pas remplie, nous décrémentons *i*. La boucle tourne jusqu'à atteindre le premier caractère de la chaîne.

Si aucune occurrence du caractère n'est trouvée, la fonction se termine en retournant la valeur NULL.

Complexité temporelle dans le pire des cas:
Dans le pire des cas où le caractère souhaité n'est pas trouvé: complexité en **O(taille(s)+C(c_strlen(s)))**

XIII-D - Implémentation

```
char * c_strrchr (const char * s, int c)
{
   const char * p = s + c_strlen (s);

   do
   {
      if (*p == (char) c)
      {
         return (char *) p;
      }
   }
   while (--p >= s);

   return NULL;
}
```

Les modifications apportées à l'implémentation ci-dessus sont mineures.

On commence par retrouver l'adresse de fin de la chaîne donc son dernier caractère, que l'on assigne au pointeur **p**. La condition de la boucle change également un peu car nous mettons l'instruction de décrémentation à l'intérieur de celle-ci. On décrémente en fait le pointeur *p* puis on teste son adresse avec celle de la chaîne *s* qui correspond donc au début de la chaîne. La boucle s'arrête lorsque le début de la chaîne à été atteint si bien sûr, aucune occurrence du caractère demandé n'a été trouvée !

XIII-E - Tests

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
   const char * str = "Ma chaine de caracteres !";
   char * p = NULL;

   p = c_strrchr (str, 'c');

   if (p != NULL)
   {
      printf ("%s\n", p);
   }

   return 0;
}
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Le test est identique à celui de la fonction précédente *c_strchr* sauf que nous recherchons la dernière occurrence du caractère passé en second argument à la fonction *c_strrchr*. Voici la sortie du test qui recherche la dernière occurrence du caractère **c**:

```
cteres !

Process returned 0 (0x0)   execution time : 0.015 s
Press any key to continue.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

XIV - c_strcmp

XIV-A - Prototype

```
int c_strcmp (const char * s1, const char * s2);
```

XIV-B - Description et comportement

La fonction *c_strcmp* compare deux chaînes de caractères, le caractère nul participe à la comparaison et par conséquent, une chaîne possédant ce caractère sera inférieure à l'autre !

La fonction renvoie trois valeurs suivant des cas précis:

- Valeur négative si la chaîne d'adresse *s1* est inférieure à la chaîne d'adresse *s2*.
- Zéro si les deux chaînes de caractères sont identiques.
- Valeur positive si la chaîne d'adresse *s1* est supérieure à la chaîne d'adresse *s2*.

La norme du C ne précise cependant pas quelle est la valeur renvoyée. Dans certaines implémentations comme sur Linux et Windows par exemple, cette fonction renvoie les valeurs -1, 0, 1 mais dans d'autres implémentations cette fonction peut directement retourner le résultat de la soustraction des deux octets différents !

*La comparaison des octets se fait par conversion du type **void** * en un type **unsigned char** * !*

XIV-C - Algorithme

Voici un algorithme possible pour la fonction *c_strcmp*:

```
algorithme
   fonction c_strcmp (s1:chaîne, s2:chaîne):entier
      debut
         i <- 0

         tant que s1[i] égal à s2[i] faire
            si s1[i] == 0
               alors retourne 0
            fsi

            i <- i + 1
         ftant

         retourne -1 si s1[i] inférieur à s2[i] sinon retourne 1
      fin

lexique
   s1 : chaîne : Chaîne de caractères pour la comparaison avec la chaîne s2.
   s2 : chaîne : Chaîne de caractères pour la comparaison.
   i  : entier : Variable de parcours des octets des chaînes.
```

L'algorithme ci-dessus parcourt les deux chaînes simultanément. L'instruction de la boucle permet de détecter la première paire d'octets qui sont différents. Le corps de la boucle permet de déterminer si l'on se trouve à la fin de

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

la chaîne *s1* où dans ce cas précis, nous mettons fin à la fonction en retournant la valeur **0** pour indiquer que les deux chaînes sont identiques.

Complexité temporelle dans le pire des cas:
Dans le pire des cas où on parcourt toute la chaîne *s1*: complexité en **O(taille(s1))**

XIV-D - Implémentation

```
int c_strcmp (const char * s1, const char * s2)
{
   const unsigned char * p_s1 = s1;
   const unsigned char * p_s2 = s2;

   while (*p_s1 == *p_s2)
   {
      if (*p_s1 == 0)
      {
         return 0;
      }

      p_s1++;
      p_s2++;
   }

   return *p_s1 < *p_s2 ? -1 : 1;
}
```

Notre implémentation est presque identique à l'algorithme à ceci près que nous utilisons des pointeurs de type **unsigned char *** (en effet, tout comme la fonction *c_memcmp*, la comparaison se fait par conversion des octets en **unsigned char**) et nous en incrémentons leur position à chaque tour de boucle ! On utilise également l'*opérateur ternaire* pour la dernière instruction **return** ce qui facilite l'implémentation de la fonction !

Si la boucle se termine avec des caractères différents, nous quittons la fonction en retournant **-1** si la chaîne *s1* est inférieure à la chaîne *s2*, **1** dans le cas contraire !

XIV-E - Tests

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
   char * str1 = "Une chaine de caracteres !";
   char * str2 = "Une chaine de caracteres !";
   char * str3 = "Une chaine... !";
   char * str4 = "Ma chaine de caracteres... !";

   printf ("Test 1 : %d\n", c_strcmp (str1, str2));
   printf ("Test 2 : %d\n", c_strcmp (str2, str3));
   printf ("Test 3 : %d\n", c_strcmp (str3, str4));

   return 0;
}
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Dans ce programme de test nous faisons trois comparaisons pour vérifier tous les points de la fonction *c_strcmp*. Nous commençons par un test d'égalité, un test d'infériorité puis pour finir un test de supériorité ! Voici ce que cela donne en sortie sur la console :

```
Test 1 : 0
Test 2 : -1
Test 3 : 1

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

XV - c_strncmp

XV-A - Prototype

```
int c_strncmp (const char * s1, const char * s2, c_size_t n);
```

XV-B - Description et comportement

La fonction *c_strncmp* compare deux chaînes de caractères sur une longueur *n*, le caractère nul participe à la comparaison et par conséquent, une chaîne possédant ce caractère sera inférieure à l'autre !

La fonction renvoie trois valeurs suivant des cas précis:

- Valeur négative si la chaîne d'adresse *s1* est inférieure à la chaîne d'adresse *s2*.
- Zéro si les deux chaînes de caractères sont identiques.
- Valeur positive si la chaîne d'adresse *s1* est supérieure à la chaîne d'adresse *s2*.

La norme du C ne précise cependant pas quelle est la valeur renvoyée. Dans certaines implémentations comme sur Linux et Windows par exemple, cette fonction renvoie les valeurs -1, 0, 1 mais dans d'autres implémentations cette fonction peut directement retourner le résultat de la soustraction des deux octets différents !

*La comparaison des octets se fait par conversion du type **void** * en un type **unsigned char** * !*

XV-C - Algorithme

Voici un algorithme possible pour la fonction *c_strncmp*:

```
algorithme
   fonction c_strcmp (s1:chaîne, s2:chaîne, n:entier):entier
      début
         i <- 0

         tant que n <- n - 1 faire
            si s1[i] égal à 0 ou s1[i] différent de s2[i] alors
               si s1[i] inférieur à s2[i]
                  alors retourne -1
                  sinon retourne 1
               fsi
            fsi

            i <- i + 1
         ftant

         retourne 0
      fin

lexique
   s1 : chaîne : Chaîne de caractères pour la comparaison avec la chaîne s2.
   s2 : chaîne : Chaîne de caractères pour la comparaison.
   n  : Longueur de la comparaison.
   i  : Variable d'incrémentation pour le parcours des adresses mémoire.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

La boucle parcourt les adresses depuis leur point de départ jusqu'à *n*. Dans la première condition nous testons si l'octet courant de la chaîne *s1* est un caractère nul ou si les deux octets courants sont différents. Si les deux octets courants sont différents, nous testons leur différence et nous retournons la valeur adéquate.

Nous pouvons voir qu'ici je n'ai pas choisi de retourner comme valeur le résultat d'une soustraction entre les deux octets mais uniquement une valeur relative au bit du signe du résultat comme le font la plupart des implémentations ! Si l'octet pointé par l'adresse *s1[i]* et inférieur à celui de l'adresse *s2[i]* nous retournons la valeur **-1**, **1** sinon.

Si la boucle se termine, c'est la valeur **0** qui est retournée car dans ce cas le contenu des deux adresses est identique !

Complexité temporelle dans le pire des cas:
Dans le pire des cas, c'est à dire une égalité parfaite: complexité en **O(taille(s1/2))**

XV-D - Implémentation

```
int c_strncmp (const char * s1, const char * s2, c_size_t n)
{
   const unsigned char * p_s1 = s1;
   const unsigned char * p_s2 = s2;

   while (n--)
   {
      if (*p_s1 == 0 || *p_s1 != *p_s2)
      {
         return *p_s1 < *p_s2 ? -1 : 1;
      }

      p_s1++;
      p_s2++;
   }

   return 0;
}
```

Comme précisé dans la description de la fonction, la comparaison se fait par conversion en type **unsigned char**, c'est ce que nous faisons dans cette implémentation en faisant pointer les deux adresses par des pointeurs de ce type !

A chaque tour de boucle nous incrémentons également les deux pointeurs *p_s1* et *p_s2* pour avancer sur l'octet suivant des deux chaînes de caractères !

XV-E - Tests

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
   char * str1 = "abcdefghij";
   char * str2 = "abcdefghij";
   char * str3 = "abcefghijk";
   char * str4 = "abbcdefghi";
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
    printf ("Test 1 : %d\n", c_strncmp (str1, str2, 4));
    printf ("Test 2 : %d\n", c_strncmp (str2, str3, 4));
    printf ("Test 3 : %d\n", c_strncmp (str3, str4, 4));

    return 0;
}
```

Dans ce programme de test nous faisons trois comparaisons pour vérifier tous les points de la fonction *c_strncmp*. Nous commençons par un test d'égalité, un test d'infériorité puis pour finir un test de supériorité ! Voici ce que cela donne en sortie sur la console :

```
Test 1 : 0
Test 2 : -1
Test 3 : 1

Process returned 0 (0x0)   execution time : 0.015 s
Press any key to continue.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

XVI - c_strpbrk

XVI-A - Prototype

```
char * c_strpbrk (const char * s1, const char * s2);
```

XVI-B - Description et comportement

La fonction *c_strpbrk* recherche la première occurrence d'un caractère de la chaîne *s2* dans la chaîne *s1* et retourne un pointeur sur le caractère trouvé.

Le zéro de fin de chaîne ne participe pas à la recherche.

La fonction renvoie la valeur NULL si aucun caractère de l'ensemble *s2* n'appartient à la chaîne *s1* !

XVI-C - Algorithme

Voici un algorithme possible pour la fonction *c_strpbrk*:

```
algorithme
   fonction c_strpbrk (s1:chaîne, s2:chaîne):chaîne
      début
         i <- 0
         j <- 0
         len <- longueur (s2)

         tant que s1[i] différent de '\0' faire
            pour j de 0 à len faire
               si s2[j] égal à s1[i]
                   alors retourne s2[j]
               fsi
            fpour

            i <- i + 1
         ftant

         retourne NULL
      fin

lexique
   s1  : chaîne : Chaîne de caractères de recherche.
   s2  : chaîne : Ensemble de caractères dont il faut trouver une occurrence dans la chaîne s1.
   i   : entier : Variable d'incrémentation pour le parcours de la chaîne s1.
   j   : entier : Variable d'incrémentation pour le parcours de l'ensemble s2.
   len : entier : Longueur de l'ensemble s2.
```

L'algorithme ci-dessus parcourt la chaîne *s1* caractère par caractère. A chaque itération, on parcours la chaîne *s2* à la recherche du caractère courant dans la chaîne *s1*.

Si le caractère courant de la chaîne *s1* correspond à un des caractères de la chaîne *s2*, on retourne l'adresse mémoire (un pointeur en C) du caractère.

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Si la première boucle se termine, on retourne NULL qui dans ce cas, indique que la chaîne *s1* ne contient pas de caractère figurant dans l'ensemble *s2* !

Complexité temporelle dans le pire des cas:
Dans le pire des cas où il n'y a aucune lettre de *s1* dans *s2*: complexité en **O(taille(s1)*taille(s2)+C(c_strlen(s2))**

XVI-D - Implémentation

```
char * c_strpbrk (const char * s1, const char * s2)
{
   const char * p_s1 = s1;
   int i = 0;
   int len = c_strlen (s2);

   while (*p_s1++)
   {
      for (i = 0; i < len; i++)
      {
         if (s2[i] == *p_s1)
         {
            return (char *)p_s1;
         }
      }
   }

   return NULL;
}
```

L'implémentation ici correspond à l'algorithme présenté ci-dessus mais est simplement adapté au C et donc à l'utilisation des pointeurs. On peut juste noter que le retour du pointeur sur le caractère trouvé (dans la condition à l'intérieur de la seconde boucle) est caster en type **char** * car effectivement, le pointeur de parcours est du type **const char** * tout comme le sont les paramètres !

XVI-E - Tests

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
   const char * s1 = "Bonjour !";
   const char * s2 = "rj";
   char * p = NULL;

   p = c_strpbrk (s1, s2);

   if (p != NULL)
   {
      printf ("%s\n", p);
   }

   return 0;
}
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Nous pouvons observer le comportement de la fonction avec ce simple programme. Notre ensemble de caractères *s2* contient deux caractères figurant dans la chaîne *s1*. On peut noter que seule la première occurrence de l'un d'eux valide la condition de la fonction et y met fin donc ici dans notre test, c'est l'adresse du caractère **j** qui est retourné:

```
jour !

Process returned 0 (0x0)   execution time : 0.046 s
Press any key to continue.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

XVII - c_strspn

XVII-A - Prototype

```
c_size_t c_strspn (const char * s1, const char * s2);
```

XVII-B - Description et comportement

La fonction *c_strspn* calcule la longueur du segment initial de la chaîne *s1* ne contenant que des caractères de l'ensemble *s2*. Son comportement est semblable à la fonction *c_strpbrk* sauf qu'elle retourne la taille du segment initial et non un pointeur !

Le caractère de fin de chaîne ne participe pas à la recherche.

La fonction renvoie la taille du segment initial ou **0** si aucun caractère du début de la chaîne *s1* ne figure dans l'ensemble *s2*.

XVII-C - Algorithme

Voici un algorithme possible pour la fonction *c_strspn*:

```
algorithme
   fonction c_strspn (s1:chaîne, s2:chaîne):entier
      début
         i <- 0
         j <- 0
         len_s1 <- longueur (s1)
         len_s2 <- longueur (s2)
         size <- 0
         quit <- 0

         pour i de 0 à len_s1 et tant que quit égal à 0 faire
            quit <- 1
            pour j de 0 à len_s2 et tant que quit égal à 1 faire
               si s2[j] égal à s1[i]
                  alors
                     quit <- 0
                     size <- size + 1
               fsi
            fpour
         fpour

         retourne size
      fin

lexique
   s1     : chaîne : Chaîne de caractères de recherche.
   s2     : chaîne : Ensemble de caractères devant se trouver dans le segment initial de la chaîne
 s1.
   i      : entier : Variable d'incrémentation pour le parcours de la chaîne s1.
   j      : entier : Variable d'incrémentation pour le parcours de l'ensemble s2.
   len_s1 : entier : Longueur de la chaîne s1.
   len_s2 : entier : Longueur de l'ensemble s2.
   size   : entier : Taille du segment initial.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
    quit    : entier : Drapeau de sortie de la fonction.
```

Nous commençons comme d'habitude par initialiser des variables, on initialise de même à la valeur **0** la variable *size* qui est celle utilisée pour stocker la taille du segment initial ainsi que la variable *quit* qui est le drapeau de sortie de la fonction.

Notre première boucle dispose de deux conditions de sortie et se termine lorsqu'une des deux devient vraie à savoir dans l'ordre:

1. Si la variable *quit* vaut **1**
2. Si la fin de la chaîne *s1* a été atteinte.

La première instruction met la variable *quit* à la valeur **1**. Ceci nous permet de sortir de la fonction si aucun caractère valide n'a été trouvé.

Nous attaquons ensuite le parcours de l'ensemble *s2* (un parcours complet de l'ensemble est effectué à chaque tour de la première boucle). Nous possédons ici également deux conditions d'arrêt de la boucle à savoir:

1. Si la fin de l'ensemble *s2* a été atteinte.
2. Si la variable *quit* vaut **0**.

Si au cours de cette boucle un caractère de l'ensemble *s2* est présent dans le segment initial de la chaîne *s1*, nous mettons la variable *quit* à la valeur **0** ce qui nous permet de continuer à la sortie de la seconde boucle le parcours de la chaîne *s1* et nous incrémentons également le compteur *size* !

Complexité temporelle dans le pire des cas:
Dans le pire des cas *s2[j] == s1[i]* seulement lorsque *j = len_s2-1*: complexité en **O(taille(s1)*taille(s2)+C(c_strlen(s1))+C(c_strlen(s2)))**

XVII-D - Implémentation

```
c_size_t c_strspn (const char * s1, const char * s2)
{
   c_size_t j = 0;
   c_size_t i = 0;
   c_size_t len_s1 = c_strlen (s1);
   c_size_t len_s2 = c_strlen (s2);
   c_size_t size = 0;
   int quit = 0;

   for (i = 0; !quit && i < len_s1; i++)
   {
      quit = 1;
      for (j = 0; quit && j < len_s2; j++)
      {
         if (s2[j] == s1[i])
         {
            quit = 0;
            size++;
         }
      }
   }

   return size;
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
}
```

L'implémentation présentée ci-dessus reflète avec exactitude l'algorithme décrit plus haut. Il faut juste noter que les variables *i* et *j* sont déclarées en type **c_size_t** pour satisfaire des futurs avertissements du compilateur car nous faisons dans le cas contraire des comparaisons entre type **int** et **unsigned int** !

*Il n'est jamais conseillé de mettre plusieurs conditions dans des boucles **for** contrairement à ce qui a été fait ici. Cela n'arrange pas toujours la bonne lisibilité et compréhension du code mais cela nous a permis dans l'implémentation ci-dessus d'écrire un code plus court et plus simple.*

XVII-E - Tests

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
   printf ("%d\n", c_strspn ("OOoo Bonjour ! ooOO", "Oo"));
   return 0;
}
```

Dans ce simple programme de test, nous recherchons la longueur du segment initial **"OOoo"** de la chaîne passée en premier argument à la fonction *c_strspn*. Le résultat est le suivant :

```
4

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

XVIII - c_strcspn

XVIII-A - Prototype

```
c_size_t c_strcspn (const char * s1, const char * s2);
```

XVIII-B - Description et comportement

La fonction *c_strcspn* calcule la longueur du segment initial de la chaîne *s1* ne contenant aucun des caractères de l'ensemble *s2*.

Le caractère de fin de chaîne ne participe pas à la recherche.

La fonction renvoie la taille du segment initial ne contenant aucun caractère de l'ensemble *s2* ou **0** dans le cas contraire.

XVIII-C - Algorithme

Voici un algorithme possible pour la fonction *c_strcspn*:

```
algorithme
   fonction c_strcspn (s1:chaîne, s2:chaîne):entier
      début
         i <- 0
         j <- 0
         len_s1 <- longueur (s1)
         len_s2 <- longueur (s2)
         size <- 0
         quit <- 0

         pour i de 0 à len_s1 et tant que quit égal à 0 faire
            pour j de 0 à len_s2 et tant que quit égal à 0 faire
               si s2[j] égal à s1[i]
                  alors quit <- 1
               fsi
            fpour

            si quit égal à 0
               alors size <- size + 1
            fsi
         fpour

         retourne size
      fin

lexique
   s1     : chaîne : Chaîne de caractères de recherche.
   s2     : chaîne : Ensemble de caractères ne devant pas se trouver dans le segment initial de la
 chaîne s1.
   i      : entier : Variable d'incrémentation pour le parcours de la chaîne s1.
   j      : entier : Variable d'incrémentation pour le parcours de l'ensemble s2.
   len_s1 : entier : Longueur de la chaîne s1.
   len_s2 : entier : Longueur de l'ensemble s2.
   size   : entier : Taille du segment initial.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
    quit    : entier : Drapeau de sortie de la fonction.
```

Nous commençons comme d'habitude par initialiser des variables, on initialise de même à la valeur **0** la variable *size* qui est celle utilisée pour stocker la taille du segment initial ainsi que la variable *quit* qui est le drapeau de sortie de la fonction.

Notre première boucle dispose de deux conditions de sortie et se termine lorsqu'une des deux devient vraie à savoir dans l'ordre:

1. Si la variable *quit* vaut **1**
2. Si la fin de la chaîne *s1* a été atteinte.

Nous attaquons ensuite le parcours de l'ensemble *s2* (un parcours complet de l'ensemble est effectué à chaque tour de la première boucle). Nous possédons ici également deux conditions d'arrêt de la boucle à savoir:

1. Si la variable *quit* vaut **1**
2. Si la fin de la chaîne *s2* a été atteinte.

Si au cours de cette boucle un caractère de l'ensemble *s2* est présent dans le segment initial de la chaîne *s1*, nous mettons la variable *quit* à la valeur **1** ce qui nous permettra de quitter la fonction car elle aura atteint sa condition de sortie.

A la fin de la seconde boucle, si la variable *quit* vaut **0** alors on incrémente la variable *size* de **1** et on continue le parcours de la chaîne *s1* !

Complexité temporelle dans le pire des cas:
Dans le pire des cas *(s2[j] != s1[i])*: complexité en **O(taille(s1)*taille(s2)+C(c_strlen(s1))+C(c_strlen(s2)))**

XVIII-D - Implémentation

```
c_size_t c_strcspn (const char * s1, const char * s2)
{
   c_size_t j = 0;
   c_size_t i = 0;
   c_size_t len_s1 = c_strlen (s1);
   c_size_t len_s2 = c_strlen (s2);
   c_size_t size = 0;
   int quit = 0;

   for (i = 0; !quit && i < len_s1; i++)
   {
      for (j = 0; !quit && j < len_s2; j++)
      {
         if (s2[j] == s1[i])
         {
            quit = 1;
         }
      }

      if (!quit)
      {
         size++;
      }
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
    }

    return size;
}
```

L'implémentation présentée ci-dessus reflète avec exactitude l'algorithme décrit plus haut. Il faut juste noter que les variables *i* et *j* sont déclarées en type **c_size_t** pour satisfaire des futurs avertissements du compilateur car nous faisons dans le cas contraire des comparaisons entre type **int** et **unsigned int** !

Il n'est jamais conseillé de mettre plusieurs conditions dans des boucles ***for*** *contrairement à ce qui a été fait ici. Cela n'arrange pas toujours la bonne lisibilité et compréhension du code mais cela nous a permis dans l'implémentation ci-dessus d'écrire un code plus court et plus simple.*

XVIII-E - Tests

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
   printf ("%d\n", c_strcspn ("Bonjour , le monde !", " "));
   return 0;
}
```

Dans notre court exemple ci-dessus, nous calculons la taille du segment initial dont les caractères n'appartiennent pas à l'ensemble passé en second argument ce qui nous donne le mot **"Bonjour"** donc une taille de **7**:

```
7

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

XIX - c_strstr

XIX-A - Prototype

```
char * c_strstr (const char * s1, const char * s2);
```

XIX-B - Description et comportement

La fonction *c_strstr* recherche la première occurrence de la sous-chaîne *s2* dans la chaîne *s1*. Le zéro de fin ne participe pas à la recherche !

La fonction retourne un pointeur sur la position du premier caractère de la sous-chaîne retrouvée ou NULL si aucune occurrence n'a été trouvée.

Si la sous-chaîne *s2* est vide (""), alors la fonction renvoie la chaîne *s1*.

XIX-C - Algorithme

Voici un algorithme possible pour la fonction *c_strstr*:

```
algorithme
   fonction c_strstr (s1:chaîne, s2:chaîne):chaîne
      début
         i <- 0
         len <- longueur (s2)

         tant que s1[i] différent de '\0' faire
            si c_memcmp (s1[i], s2, len) égal à 0
               alors retourne s1[i]
            fsi

            i <- i + 1
         ftant

         retourne NULL
      fin

lexique
   s1  : chaîne : Chaîne de caractères de recherche.
   s2  : chaîne : Sous-chaîne à retrouver dans la chaîne s1.
   i   : entier : Variable d'incrémentation pour le parcours de la chaîne s1.
   len : entier : Longueur de l'ensemble s2.
```

Nous commençons par initialiser les variables dont la variable *len* qui contiendra la longueur de la sous-chaîne *s2*. Nous pourrions en effet éviter la déclaration d'une variable mais cela coûterait un appel de fonction supplémentaire à chaque itération de notre boucle donc baisse des performances !

Nous parcourons la chaîne *s1* tant que nous n'arrivons pas à la fin de la chaîne ou jusqu'à ce que la fonction trouve une première occurrence de la sous-chaîne *s2*, ce que nous testons dans la condition à l'intérieur de la boucle.

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Si la fonction *c_memcmp* renvoie la valeur **0** alors, cela signifie que nous avons trouvé une première sous-chaîne et on retourne dans ce cas la position du premier caractère de celle-ci !

Complexité temporelle dans le pire des cas:
Dans le pire des cas *s2* n'apparaît pas dans *s1*: complexité en **O(taille(s1)*C(c_memcmp))**

XIX-D - Implémentation

```
char * c_strstr (const char * s1, const char * s2)
{
   const char * p_s1 = s1;
   c_size_t len = c_strlen (s2);

   while (*p_s1)
   {
      if (c_memcmp (p_s1, s2, len) == 0)
      {
         return (char *)p_s1;
      }

      p_s1++;
   }

   return NULL;
}
```

L'implémentation ci-dessus est identique à l'algorithme présenté plus haut sauf que nous utilisons des pointeurs ! On peut juste remarquer que le retour de la fonction dans la condition **if** se fait par cast car effectivement, notre pointeur *p_s1* tout comme les arguments de la fonction sont du type **const char *** alors que nous devons retourner le type **char *** !

XIX-E - Tests

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
   printf ("%s\n", c_strstr ("Bonjour, le monde !", "le "));
   return 0;
}
```

Dans ce simple programme d'exemple, on recherche la sous-chaîne **"le "** ce qui nous fait afficher le résultat:

```
le monde !

Process returned 0 (0x0)   execution time : 0.031 s
Press any key to continue.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

XX - c_strtok

XX-A - Prototype

```
char * c_strtok (char * s1, const char * s2);
```

XX-B - Description et comportement

La fonction *c_strtok* permet d'éclater la chaîne de caractères *s1* en plusieurs éléments lexicaux d'après la liste des délimiteurs contenus dans l'ensemble *s2*.

Le premier appel doit fournir l'adresse d'une chaîne *s1* et les appels suivants, s'ils se font sur la même chaîne, doivent se faire en passant NULL au premier argument ! L'ensemble de délimiteurs peut par contre changer à chaque appel.

La fonction renvoie un pointeur sur le début de la sous-chaîne découpée ou NULL si plus aucun délimiteur n'est trouvé.

La fonction modifie son argument s1 en y plaçant un zéro de fin de chaîne après la fin de chaque élément lexical, il convient donc de travailler sur une copie de la chaîne !

XX-C - Algorithme

Voici un algorithme possible pour la fonction *c_strtok*:

L'algorithme ci-dessous est tiré de la fonction strtok_r trouvé sur le site Koders à l'adresse : ***strtok_r.c***

```
algorithme
   fonction c_strtok (s1:chaîne, s2:chaîne):chaîne
      début
         i <- 0
         ptr <- NULL
         ret <- NULL

         /* 1. */
         si s1 égal à NULL
            alors s1 = ptr
         fsi

         /* 2. */
         tant que s1[i] différent de '\0' et c_strchr (s2, s1[i]) égal vrai faire
            i <- i + 1
         ftant

         si s1[i] égal à '\0'
            alors
               /* 3. */
               ret = NULL
            sinon
               /* 4. */
               ret = s1[i]
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
                /* 5. */
                tant que s1[i] différent de '\0' et c_strchr (s2, s1[i]) égal faux faire
                   i <- i + 1
                ftant

                /* 6. */
                si s1[i] différent de '\0'
                   alors s1[i] = '\0'
                fsi

                /* 7. */
                ptr = s1[i]
          fsi

          retourne ret
       fin

lexique
   s1  : chaîne : Chaîne de caractères à découper.
   s2  : chaîne : Ensemble de délimiteurs à considérer pour la découpe.
   i   : entier : Variable d'incrémentation pour le parcours de la chaîne s1.
   ptr : chaîne : Variable statique permettant de stocker le dernier emplacement de la découpe.
   ret : chaîne : Partie courante de la chaîne découpée. Variable retournée par la fonction.
```

L'algorithme ci-dessus est sans doute le plus long et compliqué de cet article, les explications seront donc scindées en 7 groupes par rapport à la numérotation qui est en place dans l'algorithme !

On commence comme d'habitude par déclarer et initialiser nos variables locales. Ici, la variable *ptr* nous permet de stocker le début de la prochaine découpe lors d'un nouvel appel de la fonction si bien sûr, le premier argument est mis sur la valeur NULL !

1. On commence par déterminer si le premier argument vaut NULL. Cela a une incidence sur le bon déroulement de l'éclatement de la chaîne de caractères. Si vous devez appeler cette fonction plusieurs fois sur la même chaîne, il vous faut passer cet argument avec la valeur NULL ! Ici si l'argument *s1* vaut NULL, on passe alors l'adresse contenue dans *ptr* à *s1*.
2. Cette première boucle permet de repérer le premier caractère de la partie de la chaîne à découper donc en recherchant tout simplement le premier caractère qui ne se trouve pas dans l'ensemble des délimiteurs *s2*.
3. Si après notre première recherche le caractère courant n'est autre que le zéro de fin de chaîne, on attribue la valeur NULL à la variable *ret* et on quitte la fonction.
4. Si on se trouve après notre première recherche sur le début de la sous-chaîne à renvoyer, on attribue à la variable *ret* l'adresse du début de cette dernière.
5. Cette seconde boucle permet de calculer la longueur de la sous-chaîne à renvoyer. Ceci permet à la fonction de savoir où placer le nouveau zéro de fin de chaîne. C'est en effet le cas où la fonction *c_strtok*, tout comme la version offcielle, modifie la chaîne source en y plaçant des zéros de fin de chaîne après chaque découpage, il convient donc de travailler sur des copies des chaînes sous peine de ne plus pouvoir les utiliser !
6. Si l'on ne se trouve pas sur un zéro de fin de chaîne, on en place un et on se déplace d'une case mémoire donc au début de la future découpe lors d'un prochain appel.
7. On transmet l'adresse du début de la prochaine découpe à la variable *ptr* qui est je le rappelle une variable statique, elle garde donc sa valeur même lors de la fin de la fonction !

Complexité temporelle dans le pire des cas:
Dans le pire des cas on parcourt UNE seule fois la chaine *s1*: complexité en **O(taille(s1)*C(c_strchr))**

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

XX-D - Implémentation

```
char * c_strtok (char * s1, const char * s2)
{
   static char * ptr = NULL;
   char * ret = NULL;

   if (s1 == NULL)
   {
      s1 = ptr;
   }

   while (*s1 && c_strchr (s2, *s1))
   {
      ++s1;
   }

   if (*s1 == '\0')
   {
      ret = NULL;
   }
   else
   {
      ret = s1;

      while (*s1 && !c_strchr (s2, *s1))
      {
         ++s1;
      }

      if (*s1)
      {
         *s1++ = '\0';
      }

      ptr = s1;
   }

   return ret;
}
```

Cette implémentation est la copie conforme de l'algorithme présenté plus haut mais adapté au C.

XX-E - Tests

```
#include "c_string.h"
#include <stdio.h>


int main (void)
{
   char str [] = "Ma chaine de caracteres !";
   char * p_str = str;
   char * p = NULL;

   while ((p = c_strtok (p_str, " ")) != NULL)
   {
      printf ("%s\n", p);
      p_str = NULL;
   }

   return 0;
}
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

Dans ce programme de test, on veut faire découper la chaîne pointée par *str*, avec comme délimiteur un simple espace:

```
Ma
chaine
de
caracteres
!

Process returned 0 (0x0)   execution time : 0.046 s
Press any key to continue.
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

XXI - Le code source complet de l'article

Vous pouvez télécharger l'archive du code source complet **ici**

XXI-A - Fichier: c_stddef.h

```
#ifndef _H_CSTDDEF
#define _H_CSTDDEF


#undef NULL
#define NULL ((void *) 0)


typedef unsigned int c_size_t;


#endif /* _H_CSTDDEF */
```

XXI-B - Fichier: c_string.h

```
#ifndef _H_CSTRING
#define _H_CSTRING


#include "c_stddef.h"


void     *  c_memset    (void * s, int c, c_size_t n);
void     *  c_memcpy    (void * dest, const void * src, c_size_t n);
void     *  c_memmove   (void * dest, const void * src, c_size_t n);
int         c_memcmp    (const void * s1, const void * s2, c_size_t n);
void     *  c_memchr    (const void * s, int c, c_size_t n);
c_size_t    c_strlen    (const char * s);
char     *  c_strcpy    (char * dest, const char * src);
char     *  c_strncpy   (char * dest, const char * src, c_size_t n);
char     *  c_strcat    (char * dest, const char * src);
char     *  c_strncat   (char * dest, const char * src, c_size_t n);
char     *  c_strchr    (const char * s, int c);
char     *  c_strrchr   (const char * s, int c);
int         c_strcmp    (const char * s1, const char * s2);
int         c_strncmp   (const char * s1, const char * s2, c_size_t n);
char     *  c_strpbrk   (const char * s1, const char * s2);
c_size_t    c_strspn    (const char * s1, const char * s2);
c_size_t    c_strcspn   (const char * s1, const char * s2);
char     *  c_strstr    (const char * s1, const char * s2);
char     *  c_strtok    (char * s1, const char * s2);


#endif /* _H_CSTRING */
```

XXI-C - Fichier: c_string.c

```
#include "c_string.h"


void * c_memset (void * s, int c, c_size_t n)
{
   unsigned char * p_s = s;
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
    while (n--)
    {
       *p_s++ = c;
    }

    return s;
}


void * c_memcpy (void * dest, const void * src, c_size_t n)
{
    char * p_dest = dest;
    const char * p_src = src;

    while (n--)
    {
       *p_dest++ = *p_src++;
    }

    return dest;
}


void * c_memmove (void * dest, const void * src, c_size_t n)
{
    char * p_dest = dest;
    const char * p_src = src;

    if (p_src <= p_dest)
    {
       p_dest += n - 1;
       p_src += n - 1;

       while (n--)
       {
          *p_dest-- = *p_src--;
       }
    }
    else
    {
       c_memcpy (dest, src, n);
    }

    return dest;
}


int c_memcmp (const void * s1, const void * s2, c_size_t n)
{
    const unsigned char * p_s1 = s1;
    const unsigned char * p_s2 = s2;

    while (n--)
    {
       if (*p_s1 != *p_s2)
       {
          return *p_s1 < *p_s2 ? -1 : 1;
       }

       p_s1++;
       p_s2++;
    }

    return 0;
}


void * c_memchr (const void * s, int c, c_size_t n)
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
{
   const unsigned char * p_s = s;

   while (n--)
   {
      if (*p_s == c)
      {
         return (void *) p_s;
      }

      p_s++;
   }

   return NULL;
}


c_size_t c_strlen (const char * s)
{
   c_size_t size = 0;

   while (*s++)
   {
      size++;
   }

   return size;
}


char * c_strcpy (char * dest, const char * src)
{
   while ((*dest++ = *src++));
 return dest;
}


char * c_strncpy (char * dest, const char * src, c_size_t n)
{
   c_size_t i = 0;

   if (n > 0)
   {
      for (i = 0; i < n; i++)
      {
         if (src[i] != '\0')
         {
            dest[i] = src[i];
         }
         else
         {
            break;
         }
      }

      if (i < n)
      {
         do
         {
            dest[i] = '\0';
            i++;
         }
         while (i < n);
      }
   }

   return dest;
}
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
char * c_strcat (char * dest, const char * src)
{
   const char * p1 = src;
   char * p2 = dest + c_strlen (dest);

   do
   {
      *p2++ = *p1;
   }
   while (*p1++);

   return dest;
}


char * c_strncat (char * dest, const char * src, c_size_t n)
{
   const char * p1 = src;
   char * p2 = dest + c_strlen (dest);

   do
   {
      *p2++ = *p1++;
   }
   while (n-- && *p1);

   return dest;
}


char * c_strchr (const char * s, int c)
{
   const char * p = s;

   while (*p != (char) c)
   {
      if (*p == 0)
      {
         return NULL;
      }

      p++;
   }

   return (char *) p;
}


char * c_strrchr (const char * s, int c)
{
   const char * p = s + c_strlen (s);

   do
   {
      if (*p == (char) c)
      {
         return (char *) p;
      }
   }
   while (--p >= s);

   return NULL;
}


int c_strcmp (const char * s1, const char * s2)
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
{
   const unsigned char * p_s1 = s1;
   const unsigned char * p_s2 = s2;

   while (*p_s1 == *p_s2)
   {
      if (*p_s1 == 0)
      {
         return 0;
      }

      p_s1++;
      p_s2++;
   }

   return *p_s1 < *p_s2 ? -1 : 1;
}


int c_strncmp (const char * s1, const char * s2, c_size_t n)
{
   const unsigned char * p_s1 = s1;
   const unsigned char * p_s2 = s2;

   while (n--)
   {
      if (*p_s1 == 0 || *p_s1 != *p_s2)
      {
         return *p_s1 < *p_s2 ? -1 : 1;
      }

      p_s1++;
      p_s2++;
   }

   return 0;
}


char * c_strpbrk (const char * s1, const char * s2)
{
   const char * p_s1 = s1;
   int i = 0;
   int len = c_strlen (s2);

   while (*p_s1++)
   {
      for (i = 0; i < len; i++)
      {
         if (s2[i] == *p_s1)
         {
            return (char *)p_s1;
         }
      }
   }

   return NULL;
}


c_size_t c_strspn (const char * s1, const char * s2)
{
   c_size_t j = 0;
   c_size_t i = 0;
   c_size_t len_s1 = c_strlen (s1);
   c_size_t len_s2 = c_strlen (s2);
   c_size_t size = 0;
   int quit = 0;
```

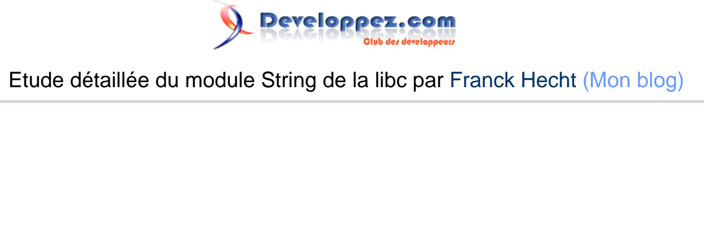

Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
   for (i = 0; !quit && i < len_s1; i++)
   {
      quit = 1;
      for (j = 0; quit && j < len_s2; j++)
      {
         if (s2[j] == s1[i])
         {
            quit = 0;
            size++;
         }
      }
   }

   return size;
}


c_size_t c_strcspn (const char * s1, const char * s2)
{
   c_size_t j = 0;
   c_size_t i = 0;
   c_size_t len_s1 = c_strlen (s1);
   c_size_t len_s2 = c_strlen (s2);
   c_size_t size = 0;
   int quit = 0;

   for (i = 0; !quit && i < len_s1; i++)
   {
      for (j = 0; !quit && j < len_s2; j++)
      {
         if (s2[j] == s1[i])
         {
            quit = 1;
         }
      }

      if (!quit)
      {
         size++;
      }
   }

   return size;
}


char * c_strstr (const char * s1, const char * s2)
{
   const char * p_s1 = s1;
   c_size_t len = c_strlen (s2);

   while (*p_s1)
   {
      if (c_memcmp (p_s1, s2, len) == 0)
      {
         return (char *)p_s1;
      }

      p_s1++;
   }

   return NULL;
}


char * c_strtok (char * s1, const char * s2)
{
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.

```
    static char * ptr = NULL;
    char * ret = NULL;

    if (s1 == NULL)
    {
       s1 = ptr;
    }

    while (*s1 && c_strchr (s2, *s1))
    {
       ++s1;
    }

    if (*s1 == '\0')
    {
       ret = NULL;
    }
    else
    {
       ret = s1;

       while (*s1 && !c_strchr (s2, *s1))
       {
          ++s1;
       }

       if (*s1)
       {
          *s1++ = '\0';
       }

       ptr = s1;
    }

    return ret;
}
```


Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.



Copyright © 2007 - Franck Hecht. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de domages et intérêts.