

Ce TP se déroule sur deux séances (quatre heures). Les étudiants qui se sentent en difficulté devraient réaliser la section 1, uniquement. Le contenu de la section 2, réservé aux étudiants plus à l'aise, n'est pas au programme de l'examen.

## 1 Réalisation d'un type de chaînes de caractères

Cette partie est l'occasion d'introduire des outils importants tels que `valgrind`, `gdb` et les *makefiles* (voir le support de cours).

**Question 1.** Le programme C suivant lit une chaîne de caractères au clavier et l'imprime sur la sortie standard.

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

int main ()
{   int c;

    c = getchar ();
    while (! isspace (c))
    {   putchar (c);
        c = getchar ();
    }
    putchar ('\n');
    return 0;
}
```

Modifier ce programme de telle sorte que tous les caractères lus soient stockés dans une variable `s`, de type `char*`, avant d'être imprimés. La variable `s` doit pointer vers une zone allouée dynamiquement, redimensionnée dès qu'elle est pleine (par exemple, en lui rajoutant huit caractères). À l'exécution, on ne veut ni débordement de tableau, ni fuite mémoire (tester avec `valgrind`).

**Question 2.** Reprendre l'exercice précédent, en appliquant une méthode de programmation modulaire. Réaliser un module `chaîne` au moyen de deux fichiers (`chaîne.c` et `chaîne.h`) ainsi qu'un fichier `main.c` contenant votre programme principal. À l'exécution, on ne veut ni débordement de tableau, ni fuite mémoire (tester avec `valgrind`) Consignes :

1. Le type doit s'appeler `struct chaîne`. Cette structure doit comporter un champ de type `char*` pointant vers une zone allouée dynamiquement (plus d'autres champs éventuels).
2. Faites le minimum : un constructeur, un destructeur, une action qui ajoute un caractère à la fin d'une chaîne existante, et une action pour imprimer la chaîne.

3. Dans le fichier d'entête, spécifier la structure. Attention, c'est plus subtil qu'on ne le croit. Faites attention à la chaîne vide.
4. Utiliser les options de compilation `-Wall` et `-Wmissing-prototypes`. Compiler les fichiers séparément.
5. Le programme principal ne doit comporter que deux variables : `s`, de type `struct chaine` et `c`, de type `int`.

**Question 3.** Reprendre l'exercice précédent en changeant l'implantation du type `struct chaine` en une liste de caractères. Partir de l'implantation des listes de `double` accessibles<sup>1</sup> par :

```
$ wget --no-cache http://www.lifl.fr/~boulrier/polycopies/SD/liste_double.tgz
$ tar xzf liste_double.tgz
```

Réaliser un module `liste_char` au moyen de deux fichiers (`liste_char.c` et `liste_char.h`). Modifier l'implantation du module `chaine`, pas les entêtes des fonctions. À l'exécution, on ne veut ni débordement de tableau, ni fuite mémoire (tester avec `valgrind`). Consignes :

1. Le type `struct chaine` doit comporter un unique champ :

```
struct chaine
{
    struct liste_char L;
};
```
2. Faut-il ajouter une directive `#include` dans `chaine.h` ?
3. Quelle implantation des listes convient le mieux à cette question ?
4. Utiliser les options de compilation `-Wall` et `-Wmissing-prototypes`. Compiler les fichiers séparément.
5. Construire un *makefile* en utilisant la méthode décrite dans le support de cours.

## 2 Le codage UTF-8

Les caractères non ASCII (les caractères accentués, en particulier), sont fréquemment une source de problèmes en informatique. En France, on utilise couramment deux codages différents pour les représenter : le codage ISO 8859-1 (appelé aussi Latin-1) et le codage UTF-8. Ce dernier prend de plus en plus d'importance, avec le développement d'internet, puisqu'il permet de coder des textes écrits dans toutes les langues terrestres<sup>2</sup>.

---

1. Ou avec un navigateur web, à partir de [www.lifl.fr/~boulrier](http://www.lifl.fr/~boulrier).

2. Plus quelques autres, comme le Klingon.

**Question 4.** Dans cette question ouverte, on vous demande de réfléchir à la conception d'un module qui aide à résoudre ce problème de cohabitation de différents formats.

Le module doit permettre de stocker, dans des variables, des chaînes passées en paramètre suivant les deux formats (UTF-8 et Latin-1), de les afficher suivant les deux formats, de déterminer le nombre d'octets<sup>3</sup> nécessaires à la représentation des chaînes sous la forme d'un tableau, ainsi que le nombre de caractères de la chaîne. Bien qu'on ne gère que deux formats, on vous demande de réfléchir à une solution facilement extensible.

Vous n'aurez probablement pas le temps de réaliser ce module intégralement. On vous demande de réaliser au moins les fichiers d'entêtes (implantation, prototype des fonctions globales, spécifications des structures de données et des fonctions). Si vous tentez de réaliser le code, supposez, dans un premier temps, que les chaînes de caractères sont correctement codées (ne vérifiez pas la validité des formats).

## Une introduction aux codages des caractères

**ASCII.** Un code ASCII est un entier codé sur 7 bits, dans l'intervalle  $[0, 0x7f = 127]$ . Parmi les caractères représentables en ASCII, on trouve : les chiffres, les lettres (non accentuées) minuscules et majuscules, des signes de ponctuations. Habituellement, un code ASCII est codé sur un octet (le bit de poids fort valant 0).

**Latin-1.** C'est le jeu de caractères par défaut sous Linux. C'est une extension de l'ASCII : tous les codes ASCII sont aussi des codes Latin-1 mais des codes supplémentaires (comportant toutes les lettres accentuées de la langue française) sont définis. Ces codes supplémentaires sont des entiers de l'intervalle<sup>4</sup>  $[0xa0, 0xff]$  (les entiers de l'intervalle  $[0x80, 0x9f]$  ne correspondent à aucun code Latin-1). Habituellement, un code Latin-1 est codé sur un octet.

**UTF-8.** C'est un des codages possibles du jeu de caractères UNICODE. En UTF-8, un caractère est codé sur 1, 2, 3 ou 4 octets. Il a une *valeur*<sup>5</sup> dans l'intervalle  $[0, 0x10ffff]$ . Un caractère codé sur 1 octet est un code ASCII. Sa valeur appartient donc à l'intervalle  $[0, 0x7f = 127]$ .

Si un caractère est codé sur 2, 3 ou 4 octets, alors chaque octet est composé d'une suite de bits de contrôle, puis de bits destinés à former la valeur du caractère, comme on l'explique

---

3. Dans tout ce qui suit, on prend bien soin de distinguer les *caractères* des *octets*.

4. Ceci explique que la fonction `getchar` retourne un `int` et pas un `char`. Si elle retournait un `char`, on ne pourrait pas distinguer le caractère `0xff` de l'entier `-1`, qui code EOF. On rencontrerait alors des fins de fichiers prématurées sur certains fichiers non ASCII. Comme elle retourne un `int`, le caractère `0xff` est retourné sous la forme `0x00ff`, qui ne peut pas être confondu avec EOF, codé par `0xffff`.

5. En ASCII ou en Latin-1, il est naturel de confondre la valeur (par exemple, `0x41 = 65` pour le 'A') et le codage (le nombre `0x41`, codé sur un octet). En UTF-8, le codage d'un caractère, à partir de sa valeur, est plus compliqué, ce qui pousse à distinguer les deux notions. Pour être vraiment précis, UNICODE est l'ensemble des valeurs possibles. UTF-8 est un codage particulier d'UNICODE (le plus populaire), mais il en existe d'autres, comme UTF-16 et UTF-32. Ça vous paraît compliqué ? Si on veut vraiment creuser les choses, ça le devient encore bien davantage et on finit par se demander : « mais qu'est-ce que c'est, un caractère ? ». Aux personnes intéressées, on conseille la lecture (passionnante) de [1].

ci-dessous.

Soit un caractère codé par  $n$  octets ( $2 \leq n \leq 4$ ). Le premier octet commence par  $n$  bits à 1, suivi d'un bit à 0, suivi de  $7 - n$  bits de valeur. Les autres octets commencent par les deux bits 10, suivi de 6 bits de valeur. La valeur du caractère s'obtient en concaténant les bits de valeur. Exemple :

2 octets.	Codage	:	110xxxxx	10yyyyyy		
	Valeur	:	xxx	xyyyyyyy		
3 octets.	Codage	:	1110xxxx	10yyyyyy	10zzzzzz	
	Valeur	:	xxxxyyyy	yyzzzzzz		
4 octets.	Codage	:	11110xxx	10yyyyyy	10zzzzzz	10wwwww
	Valeur	:	xxxxy	yyyzzzz	zzwwwww	

Certains codages sont interdits :

- le codage d'une valeur doit se faire sur un nombre minimal d'octets :
  1. les premiers octets 0xc0 et 0xc1 sont interdits (ils donneraient lieu à un code ASCII sur deux octets) ;
  2. si le premier octet est 0xe0, la valeur finale doit être supérieure ou égale à 0x800 ;
  3. si le premier octet est 0xf0, la valeur finale doit être supérieure ou égale à 0x10000.
- la valeur ne doit pas dépasser 0x10ffff ;
- les valeurs dans l'intervalle [0xd800, 0xdfff] sont réservées pour coder de l'UTF-16 et ne correspondent à aucun caractère UTF-8 ;
- il existe 66 valeurs pour des « non-caractères » : les valeurs comprises entre 0xfdd0 et 0xfdef ainsi que les 34 valeurs terminant par 0xffffe et 0xffff (de 0xffffe, 0xffff à 0x10ffffe, 0x10fff).

La conversion entre ASCII et UTF-8 est immédiate, puisque tout document ASCII est un cas particulier de document UTF-8. La conversion entre Latin-1 et UTF-8 n'est pas bien compliquée non plus : les codes ASCII sont inchangés ; les codages UTF-8 de valeur appartenant à [0xa0, 0xff] correspondent aux codages Latin-1 appartenant à ce même intervalle.

Le symbole « € » (qui ne fait pas partie du jeu de caractères Latin-1), est codé sur 3 octets : 0xe2, 0x82 et 0xac. Sa valeur est 0x20ac. Certaines lettres, utiles en Français, n'existent pas en Latin-1<sup>6</sup>, comme « œ, Œ » et même, pour certains noms propres, « ÿ ». En UTF-8, elles sont toutes codées sur deux octets.

Le *replacement character*, qui apparaît souvent sous la forme d'un point d'interrogation blanc sur fond noir, est utilisé par les routines d'impression UTF-8, pour désigner un codage non reconnu. Il est codé sur 3 octets : 0xef, 0xbf et 0xbd. Sa valeur est 0xfffd.

## Quelques commandes utiles

La commande `od` imprime les valeurs des octets d'un fichier. La commande `file` fournit le codage d'un fichier texte. La commande `iconv` convertit un fichier d'un codage dans un autre.

---

6. Mais elles existent en ISO 8859-15, appelé aussi : Latin-9.

```

$ cat fichier.txt
L titia, Fran ois,   table !
$ od -t x1 -c fichier.txt
00000000 4c c3 a6 74 69 74 69 61 2c 20 46 72 61 6e c3 a7
          L 303 246 t i t i a , F r a n 303 247
00000020 6f 69 73 2c 20 c3 a0 20 74 61 62 6c 65 20 21 0a
          o i s , 303 240 t a b l e ! \n
$ file -i fichier.txt
fichier.txt: text/plain; charset=utf-8
$ iconv --from utf-8 --to iso8859-1 < fichier.txt > fichier2.txt
$ cat fichier2.txt
L titia, Fran ois, table !
$ od -t x1 -c fichier2.txt
00000000 4c e6 74 69 74 69 61 2c 20 46 72 61 6e e7 6f 69
          L 346 t i t i a , F r a n 347 o i
00000020 73 2c 20 e0 20 74 61 62 6c 65 20 21 0a
          s , 340 t a b l e ! \n
00000035
$ file -i fichier2.txt
fichier2.txt: text/plain; charset=iso-8859-1

```

**Exercice.** Les commandes ci-dessus ont   t   ex  cut  es dans un terminal, qui attend des cha  nes de caract  res suivant un certain format.    votre avis, lequel ?

## Manipulations de bits en C

Des op  rateurs utiles sont : `&` (le « et » bit    bit), `|` (le « ou inclusif » bit    bit), `^` (le « ou exclusif » bit    bit). On peut leur ajouter les op  rateurs `<<` (d  calage des bits    gauche, d’un certain nombre de positions) et `>>` (d  calage    droite), dont on pourrait se passer, puisqu’ils peuvent se traduire par des multiplications et des divisions par des puissances de 2.

Attention au fait qu’un `char`, en C, est en r  alit  , le type « entier sign   sur 8 bits » (dans l’intervalle  $[-128, 127]$ , donc). Lorsqu’on manipule des caract  res non ASCII, il est souvent pr  f  rable d’utiliser le type `unsigned char` du C, dont les valeurs appartiennent     $[0, 255]$ . Lors des fr  quentes conversions en `int`, on   vite ainsi la propagation du bit de signe (bit num  ro 7) sur les octets de poids fort.

L’action suivante analyse et d  compose le premier octet d’un caract  re UTF-8.

```

/*
 * D  compose l'octet c (cens     tre le premier octet d'une s  quence UTF-8).
 * L'entier *counter re  oit le nombre de bits de contr  le    1
 * Le long *code re  oit les bits de valeur.
 *
 * Par exemple, si c = 110xxxxx alors *counter = 2 et *code = xxxxx.
 * Si c = 0xxxxxxx (code ASCII) alors *counter = 0 et *code = c.
 */

static void analyse_premier_octet (int* counter, long* code, unsigned char c)
{
    int cpt, bits, mask;

```

```

    cpt = 0;
    bits = c;
    mask = 0x80;                /* 1000 0000 binaire */
/*
* Invariants de boucle :
* - mask comporte un seul bit à 1
* - bits est égal à c, sans les bits à 1, situés plus à gauche que mask
* - cpt est égal au nombre de bits à 1 de c, situés plus à gauche que mask
*/
    while ((bits & mask) != 0) /* tq bits comporte un bit à 1 face à mask */
    {
        bits ^= mask;        /* effacer ce bit */
        cpt += 1;
        mask >>= 1;          /* décaler d'une position vers la droite
                               l'unique bit à 1 de mask */
    }
    *counter = cpt;
    *code = (long)bits;
}

```

## Références

- [1] Jukka Korpela. A tutorial on character code issues. <http://www.cs.tut.fi/~jkorpela/chars.html>, 2001.