Langage C
Les chaînes de caractères - version 0.03

1 Les caractères

Quand nous écrivons

```
1 char c = 'f';
```

la variable `c` contient en fait le code du caractère `'f'`. Le code d'un caractère est un nombre entier, par exemple le code ASCII de `'f'` est 102 et si on écrit

```
1 char x = 102;
```

les variables `c` et `x` contiennent la même valeur.
Le code universellement utilisé est le code ASCII qui associe un nombre de 0 à 127 à chaque de 128 caractères couverts par ce codage. Vous trouverez facilement le tableau de code ASCII sur internet, par exemple
`https://fr.wikibooks.org/wiki/Les_ASCII_de_0_à_127/La_table_ASCII`
Le code ASCII permet le codage de très peu de caractères : par exemple, seules les lettres de l'alphabet latin sans accents sont représentées dans le code ASCII. Connaître les codes de différents caractères est sans intérêt pour nous.
Ce qu'il faut savoir c'est qu'on a toujours `sizeof(char) == 1`, c'est-à-dire, tous les caractères sont codés sur un octet.
Quand on compare des caractères, on compare en fait les codes de ces caractères, par exemple la condition `'a'<= c && c <= 'z'` où `c` est une variable de type `char` est satisfaite si et seulement si `c` contient (le code d') une lettre minuscule. Nous pouvons le voir en regardant le tableau des codes ASCII où nous constatons que tous les entiers entre 97 (le code de `'a'`) et 122 (le code de `'z'`) sont les codes des lettres minuscules. Il est même possible de faire l'arithmétique,

```
1 char c = 'a';
2 char y = c + 2;
3 c++;
```

Après l'exécution de ce fragment de code, `c` contient (le code de) la lettre `'b'` et `y` contient (le code de) `'c'`. Le résultat dépend ici du codage ASCII, et il vaut mieux écrire des programmes qui ne dépendent pas des propriétés de ces codages (le langage C suppose que les caractères occupent 1 octet, mais pas forcement que le codage utilisé est l'ASCII).
Le langage C ne dit pas si un `char` vu comme une valeur entière est un entier signé ou non signé : `char` peut être synonyme de `signed char` sur certains machines et `unsigned char` sur d'autres. Notez qu'un octet permet de stocker 255 valeurs entières différentes : il est donc possible de mettre dans une variable `char` une valeur entière qui n'est pas le code ASCII d'un caractère (il y a seulement 128 caractères ASCII). Le fichier `limits.h` contient les constantes symboliques `CHAR_MAX` et `CHAR_MIN` dont les valeurs sont respectivement l'entier maximal et l'entier minimal qu'on peut stocker dans un `char`.

Certains caractères sont représentés par des constantes composées de deux caractères, par exemple `'\n'` – le caractère de nouvelle ligne, ou `'\t'` – le caractère de tabulation. Dans les deux cas cela représente toujours un caractère stocké dans un seul octet.
Notez aussi que les caractères représentant les chiffres, par exemple `'0'`, `'1'`, `'3'`, ..., ne sont pas codés par les nombres 0, 1, 2, etc. Par exemple le code ASCII du caractère `'0'` représentant le chiffre zéro est 48.
Finalement, parmi les caractères codés, il y a celui qu'on appelle le caractère nul, représenté en C par `'\0'`. Le code de ce caractère est 0.
Le C standard fournit plusieurs fonctions qui permettent de tester ou transformer les caractères. Les prototypes de ces fonctions se trouvent dans le fichier `ctype.h`.
Les fonctions suivantes retournent 0 si le test correspondant échoue et une valeur différente de 0 quand le test réussit (par exemple `isalpha('+')` retourne 0 puisque `'+'` n'est pas une lettre).

`int isalnum(int)`	lettre ou chiffre
`int isalpha(int)`	lettre
`int isascii(int)`	caractère ascii
`int isblank(int)`	`'\t'` ou `' '` et d'autres caractères blancs (dépend de la langue locale)
`int isdigit(int)`	un chiffre décimal de `'0'` à `'9'`
`int islower(int)`	lettre minuscule
`int isupper(int)`	lettre majuscule
`int isspace(int)`	un de caractères `'\t'`, `'\n'`, `'\v'`, `'\f'`, `'\r'`, `' '`
`int isxdigit(int)`	chiffre hexadécimal
`int ispunct(int)`	caractères imprimables, sauf l'espace, les lettres et chiffres

Les fonctions suivantes retournent respectivement une lettre majuscule (minuscule) correspondante :

`int toupper(int)`	exemple `toupper('a') == 'A'`
`int tolower(int)`	exemple `tolower('B') == 'b'`

Si l'argument de `toupper` (`tolower`) n'est pas une lettre minuscule (majuscule) la fonction retournera simplement son argument.
Si vous vous demandez pourquoi l'argument de toutes ces fonctions est un `int` et non pas un `char`, et pourqoi `toupper` et `tolower` retournent un `int` au lieu de `char`, il y a une raison technique à cela. Dans la pratique, on peut l'ignorer et utiliser les `char` comme paramètres.

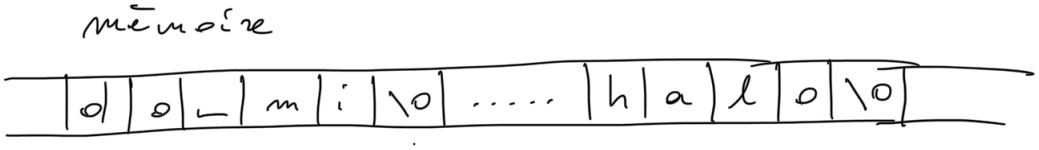
2 Chaînes de caractères (strings)

En C, une chaîne de caractères est une suite de caractères (ou plus exactement de codes de caractères) qui se termine par le caractère nul `'\0'`. Le caractère nul marque la fin de la chaîne, mais il n'est pas considéré comme faisant partie de la chaîne. Cela implique que le caractère nul ne peut jamais appartenir à une chaîne de caractères, il s'agit juste d'un marqueur à la fin de la chaîne. Une suite de caractères qui ne se termine pas par le caractère nul n'est pas une chaîne de caractères dans le sens du langage C.
Le langage C ne possède pas de type spécial pour les chaînes de caractères. Dans le programme on accède à une chaîne de caractère via un pointeur `char *` vers le premier caractère de la chaîne.

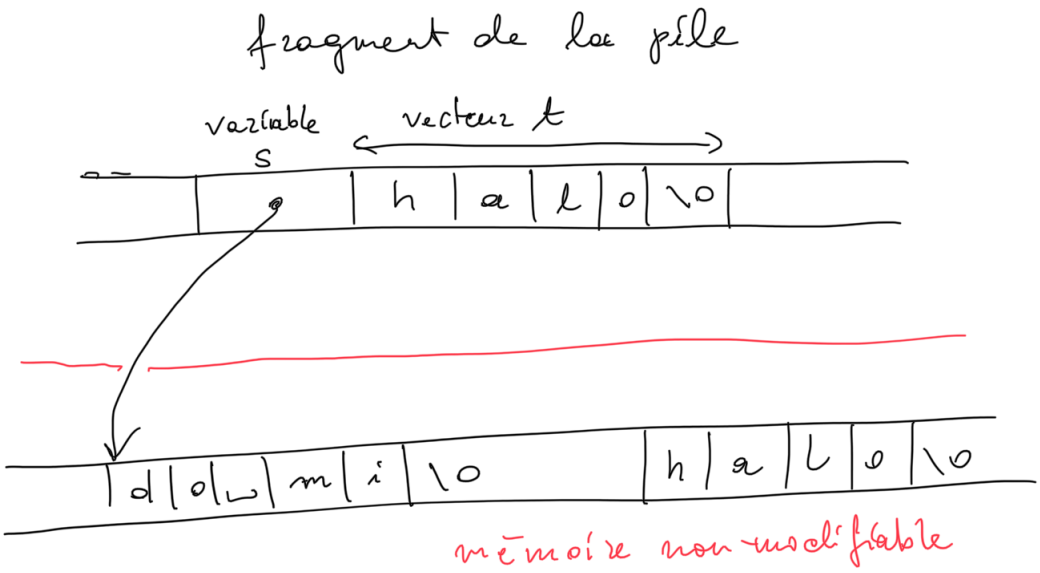
Pour simplifier, nous dirons qu'un pointeur pointe vers une chaîne de caractères quand il contient l'adresse du premier caractère de cette chaîne.
Exemple :

```
1 char *s  = "do mi";
2 char t[] = "halo";
```

Quand le compilateur rencontre des chaînes de caractères dans le code du programme il réserve la mémoire pour stocker les caractères de la chaîne, y compris le caractère nul après le dernier caractère.
Dans cet exemple le compilateur stocke quelque part dans la mémoire les chaînes `"do mi"` et `"halo"`.



Il faut note que les deux chaînes sont placées par le compilateur dans une zone de mémoire non-modifiable, c'est-à-dire accessible uniquement en lecture. Mais les deux chaînes sont utilisées de façon très différente dans `char *s = "do mi";` et `char t[] = "halo";`.
Supposons que ce fragment de code fasse partie d'une fonction. Au moment de l'appel de la fonction, la variable `s` et le vecteur `t` sont placés sur la pile.
Dans `char *s = "do mi";` la variable `s` est initialisée avec l'adresse de la chaîne non-modifiable `"do mi"`.
Par contre pour initialiser le vecteur `t` les caractères de la chaîne non-modifiable `"halo"`, y compris le caractères nul, sont copiés dans le vecteur `t`. A partir de ce moment le vecteur `t` et la chaîne non-modifiable original `"helo"` n'ont plus rien à voir un avec l'autre.
Sur le dessin ci-dessous on voit clairement que `s` pointe vers la chaîne non-modifiable originale, par contre `t` contient juste une copie d'une chaîne non-modifiable `"helo"`. D'ailleurs nous n'avons pas d'accès à la chaîne originale `"helo"` parce que nous ne connaissons pas son adresse.



Tous cela implique que les caractères pointés par `s` ne peuvent pas être modifiés. Par exemple une tentative de remplacer le premier caractère de la chaîne pointé par `s` par le caractère `'Z'` :

```
s[0] = 'Z';
```

provoque l'interruption de programme (le programme termine).
Par contre le vecteur `t`, initialisé en copiant la chaîne `"halo"`, est parfaitement modifiable.
Après initialisation le vecteur `t` contient 5 éléments : `t[0]=='h'`, `t[1]=='a'`, `t[2]=='l'`, `t[3]=='o'`, `t[4]=='\0'`.
Notez que le caractère nul est aussi copié dans `t`
Comme le vecteur `t` est sur la pile et ses éléments sont modifiables, par exemple

```
t[0]='H';
```

remplace le caractère `'h'` par `'H'`.

Pour mettre une chaîne de caractères dans un vecteur à sa déclaration, il y a deux possibilités :

```
char t[] = "abdc";
char q[] = { 'a', 'b', 'd', 'c', '\0'};
```

Les vecteurs `t` et `q` ont le même nombre d'éléments (5), et contiennent les mêmes caractères.

Revenons à notre exemple :

```
char *s = "do mi";
char t[] = "halo";

char *x = s + 3;
char *y = &s[1];
```

```
6 char *z = &t[2];
7 char *u = t + 1;
```

Après l'exécution de ce code les variables `x`, `y`, `z`, `u` contiennent aussi des pointeurs vers des chaînes de caractères,

- `x` pointe vers la chaîne composée de `mi`,
- `y` pointe vers la chaîne composée de `o␣mi`, (rappelons que `s[1]` c'est le caractère à l'indice 1 de `s`, et `&s[1]` est l'adresse de ce caractère. Notons que l'adresse `&s[1]` c'est la même chose que `s+1` donc l'instruction sur ligne peut être écrite comme `char *y = s+1;`,
- `z` contient l'adresse de la chaîne `lo`,
- `u` contient l'adresse de la chaîne composée de `alo`.

Notons aussi que l'on peut avoir un pointeur `char *` bien initialisé, mais qui ne pointe pas vers une chaîne de caractères, comme par exemple dans

```
1 char t[]={ 'a', 'b', 'c', 'd' };
2 char *s = t;
3 char *u = &t[2];
```

Ni la suite de caractères pointée par `s` ni la suite pointé par `u` ne sont pas de chaînes de caractères, puisque il n y a pas de `'\0'` pour terminer la chaîne.

2.1 La terminologie

Par abus de langage, dans les livres sur le langage C et aussi dans les pages `man` français, c'est le pointeur `char *` qui contient l'adresse d'une chaîne qui est appelé « chaîne de caractères ». Il n'y a pas trop de risque de confusion, puisque que de toute façon une chaîne de caractères est accessible uniquement via un pointeur vers le premier caractère de cette chaîne [1].

Dans ce document, je distinguerai toujours [2] la chaîne de caractère et le pointeur vers celle-ci (pointeur vers le premier caractère de la chaîne). De plus, puisqu'il est pénible d'écrire chaque fois « chaîne de caractère », très souvent j'écrirai *string*. Donc *string* est une suite de caractères qui se termine par le caractère nul. Et un pointeur vers un string est un pointeur `char *` qui contient l'adresse du premier caractère de ce string.

3 Quelques fonctions de la bibliothèque standard dont un paramètre est un pointeur vers un string

Nous présenterons les quelques fonctions les plus utilisées du C standard dont un de paramètres pointe vers un string. Pour les utiliser, il faut :

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

1. En anglais on parle soit d'un *string* soit d'un *nul-terminated string* pour dire que `char *` est l'adresse d'une chaîne de caractères.

2. Sauf oubli de ma part. Disons plutôt que je vais essayer de distinguer les chaînes de caractères et leurs adresses.

Avant d'expliquer le comportement de ces fonctions notons que certains paramètres sont de type `const char *` et d'autre juste `char *`.
`const char *s` indique que la fonction ne modifie pas le string d'adresse `s`.
Par contre, quand le paramètre est sans `const`, la fonction peut modifier le string pointé par le paramètre.

3.1 `strlen` - la longueur

```
size_t strlen( const char *s )
```

retourne la longueur du string pointé par `s`. Le caractère nul à la fin n'est pas compté, par exemple `strlen("abc")` retourne 3. Il est facile d'écrire notre propre fonction qui calcule la longueur d'un string et retourne le même résultat que la fonction `strlen` de la bibliothèque standard :

```
size_t longueur(const char *s){
  size_t len = 0;
  while(*s != '\0'){
    s++;
    len++;
  }
  return len;
}
```

Il faut souligner deux choses. Les fonctions qui opèrent sur les string n'ont aucun de moyen de vérifier si leur paramètre est bien l'adresse d'un string (qui doit toujours se terminer par `'\0'`). Par exemple `strlen` suppose qu'à l'adresse `s` commence une suite de caractères qui se termine par le caractère nul, et parcourt les caractères un à un à partir de l'adresse `s` pour trouver la fin de ce string.
D'autre part, toutes les fonctions qui prennent comme paramètre l'adresse d'un string sont obligées de parcourir tous les caractères à partir de l'adresse donnée en paramètre jusqu'au caractère nul pour découvrir où le string se termine.
Exemple. L'exemple montre le résultat de l'application de `strlen` aux différents pointeurs `char *` :

```
char *s = "abcdef";
char  v[] = "rstuwxyz";
size_t i = strlen( s ); // i == 6
i = strlen( s + 2 );    // i == 4
i = strlen( &s[3] )     // i == 3
i = strlen( v );        // i == 8
i = strlen( &v[2] )     // i == 6
i = strlen( s + 6 )     // i == 0, s+6 est l'adresse de caractère nul qui
                        // termine la chaîne pointé par s
```

3.2 `strcpy` et `strncpy` - copier un string

```
char *strcpy(char *dest, const char *src)
char *strncpy(char *dest, const char *src, size_t len)
```

`strcpy` copie le string pointé par `src` vers l'adresse `dest`. La fonction (comme toutes les fonctions de cette section) **n'alloue pas de mémoire**. Donc `dest` doit pointer vers une zone mémoire suffisamment grande pour stocker tous les caractères de `src`, y compris le caractère nul.

Exemple. Fabriquer une copie d'un string. Notez +1 dans la ligne 2 pour avoir de la place pour le caractère nul qui termine le string.

```
1 char *s = "un deux trois";
2 char *t = malloc( strlen(s) + 1 );
3 strcpy(t, s);
```

`strncpy` est la version sécurisé de `strcpy`. Dans le paramètre `len` on passe le nombre d'octets de mémoire à l'adresse `dest`.

`strncpy` copie au plus `min(len,strlen(s))` caractères de l'adresse `src` vers l'adresse `dest`. Si le nombre de caractères copiés est strictement inférieur à `len` la fonction ajoute à la fin le caractère nul.

Notez que dans certains cas le résultat à l'adresse `dest` n'est pas un string puisque il manquera le caractère nul à la fin.

`strncpy` est une version sécurisé parce que cette fonction ne copie pas au-delà de la mémoire disponible à l'adresse `dest`.

Les fonctions `strcpy` et `strncpy` retournent le premier argument.

3.2.1 `strcat` et `strncat` - concatener deux strings

```
1 char * strcat( char *dest, const char *src )
2 char * strncat(char *dest, const char *src, size_t n)
```

`strcat` copie les caractères depuis l'adresse `src`, y compris le caractère nul, en les ajoutant à la fin du string pointé par `dest`.

La mémoire à l'adresse `dest` doit être suffisamment grande pour stocker les nouveaux caractères. Cette fonction peut conduire à un débordement de mémoire ; la fonction `strncat` est préférable.

Pour comprendre ce que fait `strcat` écrivons notre version de cette fonction qui fait exactement la même chose que `strcat` :

```
1  char *mon_strcat(char *dest, const char *src){
2    char *c;
3    for( c = dest;  *c != '\0' ; c++ )
4       ;/* boucle vide */
5
6    while( *src != '\0' ){
7      *c = *src;
8      c++;
9      src++;
10   }
11   *c = '\0';
12   return dest;
13 }
```

Exemple. Le code suivant est incorrect :

```
1 char *s = "first";
2 char *t = "last";
3 strcat(s, t); //débordement à l'adresse s
```

A l'adresse `s` il n'y a de place pour les 5 nouveaux caractères : `last\0`. (De plus, `s` pointe vers un string qui réside dans la mémoire non-modifiable.)
Version correcte :

```
1 char *s="first";
2 char *t="last";
3 //n'oubliez pas +1 pour l'octet avec null
4 char *res = malloc( strlen(s) + strlen(t) + 1);
5 strcpy(res, s); //copier d'abord s à l'adresse res
6 strcat(res, t); //ajouter t dans le string pointé par res
```

En exécutant ce code on obtient `res` qui contient l'adresse de la chaîne `"firstlast"`.

Une autre version :

```
1 char *s="first";
2 char *t="last";
3 char *res = malloc( strlen(s) + strlen(t) + 1);
4 res[0] = '\0'; //res contient maintenant l'adresse de string  vide
5                // *res = '\0' ; est équivalent au res[0] = '\0';
6 strcat(res, s); //copier s à l'adresse res
7 strcat(res, t); //copier t à la fin
```

Notez que `res[0] = '\0';` à la ligne 4 est indispensable, le premier paramètre de `strcat` doit pointer vers un string.

`strncat(char *dest, char *src, size_t n)` copie `min(n, strlen(src))` caractères de l'adresse `src` vers la fin de string pointé par `dest`. A la fin `strncat` ajoute toujours en plus le caractère nul.

3.2.2 `strcmp` - comparer deux strings

```
1 int    strcmp( const char *s1, const char *s2)
2 int    strncmp(const char *s1, const char *s2, size_t n)
```

`strcmp` compare les strings pointés par `s1` et `s2` et retourne
- une valeur < 0 si le string pointé par `s1` précède le string pointé par `s2` pour l'ordre du dictionnaire (ordre lexicographique),
- 0 si les strings pointés par `s1` et `s2` sont identiques,
- une valeur > 0 si le string à l'adresse `s1` est plus grand que le string à l'adresse `s2` pour l'ordre du dictionnaire.

`strncmp` fonctionne comme `strcmp`, mais compare au maximum les `n` premiers caractères de `s1` et `s2`. En particulier, si les deux strings ont au moins `n` caractères, et que ces `n` premiers caractères sont identiques, alors `strncmp` retourne 0.
Une implémentation possible de `strcmp` :

```c
int mon_strcmp(const char *s, const char *t){
     while( *s != '\0' && *t != '\0' && *s == *t){
       s++;
       t++;
     }
     if( *s < *t )
       return -1;
     if( *s == *t)
       return 0;
     return 1;
}
```

Avertissement. Même si cela devrait être évident, il y a toujours certains (trop nombreux) qui écrivent pendant l'examen :

```c
char *s;
char *t;
/* le code qui place dans s et t les adresses de strings */
......
if( s == t ){ //égalité des pointeurs et non pas des strings
              // pointés pas s et t

}
```

en pensant que la condition `s == t` permet de savoir si les strings pointés par `s` et `t` sont égaux. Mais la condition teste seulement si les deux adresses sont égales.

Exemple. Un exemple d'utilisation de `strcmp` pour trouver l'indice d'un string maximal pour l'ordre de dictionnaire dans un vecteur de strings (voir la section 4).

```c
char *tab[]={"Julien", "Mathieu", "Marc", "Monique",
  "Jerome", "Thomas", "Alice", "Jonathan", "Barbara"};

size_t len = sizeof(tab)/sizeof(tab[0]);

size_t m = 0;
for( size_t i=1; i < len; i++ ){
  if( strcmp( tab[i], tab[m] ) > 0 )
      m = i;
}
// tab[m] la pointeur vers le string recherché
```

3.2.3 `strtok` - découper un string

```c
char * strtok( char *str, const char *sep)
```

`strtok` permet de découper le string `str` en lexèmes (tokens). Notez que `str` sera modifié par `strtok` (pas de `const` devant le premier paramètre).
On suppose que le lexème est la plus longue suite de caractères qui ne contient aucun caractère qui se trouve dans `sep`. Intuitivement, `sep` sert à spécifier les caractères qui séparent les lexèmes.

Le string `str` à découper est passé comme paramètre au premier appel à `strtok` et la fonction retournera l'adresse du premier lexème.
Pour trouver les lexèmes suivants, on appelle la fonction en boucle avec le premier paramètre NULL. Chaque appel retourne l'adresse de lexème suivant.
Quand un appel à `strtok` ne trouve plus de lexèmes, la fonction retourne `NULL`, ce qui permet de terminer la boucle.
Le paramètre `sep` peut être différent d'un appel à l'autre.
Exemple :

```
char *s = " plus grand que 0.  Pour les appels suivants  ";
//espace et point comme sepatateur
char *sep=" .";
char *lex = strtok(s, sep);
if(lex != NULL){
  printf("%s\n",lex);
}
while( ( lex = strtok(NULL, sep) ) != NULL ){
    printf("%s\n",lex);
}
```

sort sur le terminal :

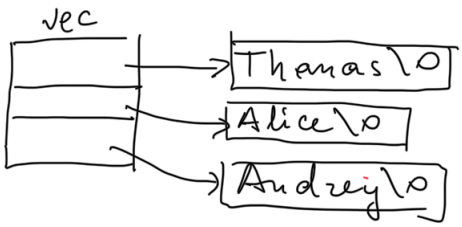
```
plus
grand
que
0
Pour
les
appels
suivants
```

4 Vecteur de strings

Ce qu'on appelle couramment « vecteur de strings » est en fait un vecteur de pointeurs vers des strings. Donc ce vecteur ne contient pas de caractères mais les adresses (les pointeurs).
Exemple :

```
char *vec[] = { "Thomas", "Alice", "Audrey"};
```

Le vecteur `vec` contient trois éléments : `vec[0]` est un pointeur vers le string `"Thomas"`, `vec[1]` un pointeur vers le string `"Alice"`, et `vec[2]` un pointeur vers le string `"Audrey"`.



Notez que les trois strings `"Thomas"`, `"Alice"`, `"Audrey"` apparaissent dans le texte du programme donc résident tous les trois dans la zone de mémoire non-modifiable, c'est-à-dire les trois pointeurs `vec[0]`, `vec[1]`, `vec[2]` pointent vers les strings accessibles uniquement en lecture, cf. section 2.

Vecteur de strings. Il devient pénible d'écrire et de lire les phrases comme : « `vec` est un vecteur d'adresses de strings ». Donc je vais plutôt utiliser la terminologie traditionnelle et parler de vecteurs de strings. Mais il ne faut jamais oublier qu'un vecteur de strings c'est toujours un vecteur de pointeurs vers des strings.

5 Paramètres de `main`

La fonction principale `main` a été utilisée jusqu'à maintenant dans sa version sans paramètres. Mais il existe aussi une version de `main` avec des paramètres, ce qui permet de traiter les paramètres de la commande exécutée sur le terminal :

```
int main(int argc, char *argv[])
```

Le paramètre `argv` est un vecteur de strings (voir la section précédente). Ce vecteur possède `argc + 1` éléments, le dernier élément à l'indice `argc` est toujours `NULL`. Tous les autres éléments de `argv` pointent vers des strings (sont différents de `NULL`).
Rappelons que si un paramètre d'une fonction a une forme de vecteur, en réalité c'est un pointeur vers le premier élément d'un vecteur. Donc `argv` est *de facto* un pointeur vers un objet de type `char *` and nous pouvons écrire de façon équivalente :

```
int main(int argc, char **argv)
```

Pour comprendre les contenu de `argv` compilons le programme suivant :

```
/* parmain.c */
#include <stdio.h>
int main(int argc, char *argv[]){
  for( int i = 0; i < argc ; i++ ){
    printf("argv[%d]=%s\n", argv[i] );
  }
}
```

On exécute l'exécutable `parmain` obtenu sur un terminal mais en ajoutant des « options » comme pour des commandes UNIX :

```
./parmain␣aaa␣12␣-o␣file
```

Le programme affichera sur le terminal

```
argv[0]=./parmain
argv[1]=aaa
argv[2]=12
argv[3]=-o
argv[4]=file
```

Nous pouvons voir que `argv[0]` pointe vers le string `"./parmain"`, c'est-à-dire vers le nom de la commande[3] avec tout le chemin que nous avons écrit sur le terminal. `argv[1]` contient le pointeur vers la chaîne de caractères `"aaa"`, `argv[2]` contient le pointeur vers la chaîne de caractères `"12"`, `argv[3]` contient le pointeur vers la chaîne de caractères `"-o"`, `argv[4]` contient le pointeur vers la chaîne de caractères `"file"`. Finalement `argv[5] == NULL`.

6 Conversion de string en nombre

```c
#include <stdlib.h>
double   atof( const char *s ); // convertit s en double
int      atoi( const char *s ); // convertit s en int
long     atol( const char *s ); // convertit s en long
```

Ces fonctions convertissent en nombre le fragment initial de string pointé par `s`. Les caractères reconnus comme « espaces » par la fonction `isspace` au début de string `s` sont ignorés.
Exemple. `atof(" \n \t -12.3abc")` retourne -12.3.
`atoi(" \n \t -12.3abc")` retourne -12.

7 Décoder un string avec `sscanf`

```c
#include <stdio.h>
int sscanf(const char *s, const char *format, ... )
```

La fonction `sscanf` permet le décodage et le découpage plus sophistiqué du string pointé par `s`. Le paramètre `format` est un format qui défini les règles de découpage.
Les paramètres qu'on met à la place ... sont de pointeurs vers des variables.
Exemple.

```c
char *s = "  12   -17   ";
int a,b;
sscanf( s, "%d%d", &a, &b);
```

Dans cet exemple les variables `a` et `b` prendront les valeurs respectivement 12 et -17.

3. J'appelle ici *commande* le programme C compilé. Les commandes UNIX, comme `ls`, sont en fait des programmes C compilés.

8 Conversion vers un string

La fonction

```
#include <stdio.h>
int snprintf( char *s, size_t size, const char *format, ...)
```

est d'une certaine façon inverse de `sscanf`. Elle transforme les valeurs d'expressions qu'on met à la place de ... en un string qui sera placé à l'adresse `s`.
`s` doit être une adresse valable qui pointe vers la mémoire qui contient au moins `size` octets (la fonction ne fait pas d'allocation de mémoire).
La fonction écrit au plus `size-1` caractères à l'adresse `s` et ajoute le caractère nul à la fin.

```
int a = -65;
double d = 4.31;
unsigned u = 120;
#define LEN 64
char tab[LEN];
snprintf( tab , LEN, "%d  %4.2f %u", a,d,u);
```

Le `snprintf` met dans `tab` le string `"-65 4.31 120"`, c'est-à-dire `tab[0]=='-', tab[1]=='6', tab[2]=='5', tab[3]=='␣'`, etc.

Pour utiliser `sscanf` et `snprintf` il faut maîtriser les formats. Nous étudierons un peu les formats avec les fonction d'entrée/sortie formatés. Mais on n'attend pas que vous soyez des experts des formats. Si à l'examen vous devez utiliser une fonction avec un format ce format vous sera donné.