Programmation C et Système
Les fonctions de `<string.h>`

Régis Barbanchon

L2 Informatique

Les fonctions de <string.h>

L'entête <string.h> déclare deux types de fonctions :

- celles en str...() opèrent sur les chaînes de caractères, c-à-d sur les tableaux de char terminés par un '\0').
- celles en mem...() opèrent sur les blocs mémoire.

Exemples (copie de chaîne et copie de bloc mémoire) :

```
char * strcpy (char * target, char const * source);
void * memcpy (void * target, void const * source, size_t n);
```

- Comme strcpy() manipule des chaînes de caractères, ses paramètres sont de type char * (pointeur sur caractère), ou char const * (pointeur sur caractère constant) lorsque le paramètre est utilisé en lecture seule.
- Comme memcpy() manipule des données de type inconnu, elle utilise le type void * (pointeur sur type inconnu), ou void const * (pointeur sur type inconnu constant) lorsque le paramètre est utilisé en lecture seule.

Les fonctions de `<string.h>` sur les chaînes

Les principales fonctions sur les chaînes de caractères sont :

- `strlen()` : calcule la longueur d'une chaîne.
- `strcmp()/strncmp()` : compare deux chaînes.
- `strcoll()` : compare deux chaînes selon la *locale*.
- `strchr()` : recherche la première occurrence d'un caractère.
- `strrchr()` : recherche la dernière occurrence d'un caractère.
- `strpbrk()` : recherche la première occ d'un ensemble de car.
- `strstr()` : recherche la première occurrence d'une chaîne.
- `strcpy()/strncpy()` : recopient une chaîne dans une autre.
- `strcat()/strncat()` : concatènent une chaîne à une autre.
- `strspn()/strcspn()` : calculent la longueur d'un préfixe.

Les fonctions de `<string.h>` sur les blocs mémoire

Les fonctions sur les blocs mémoire sont :

- `memcmp()` : compare les bytes de deux blocs.
- `memchr()` : recherche un byte dans un bloc.
- `memset()` : remplit un bloc avec la valeur d'un byte.
- `memcpy()` : recopie les bytes d'un bloc vers un autre bloc.
- `memmove()` : idem, mais les blocs peuvent se chevaucher.

Ces fonctions sont sans relation avec les chaînes.
Elles sont probablement incluses dans `<string.h>` à cause des ressemblances avec les fonctions `strcmp()`, `strchr()`, `strcpy()`.

Les prototypes des fonctions de <string.h>

Les prototypes des fonctions en str...() sur les chaînes :

```
size_t strlen (char const * string);

int strcmp  (char const * string1, char const * string2);
int strncmp (char const * string1, char const * string2, size_t n);

char * strchr  (char const * string, int character);
char * strrchr (char const * string, int character);
char * strpbrk (char const * string, char const * accept);
char * strstr  (char const * string, char const * substring);

char * strcpy  (char * target, char const * source);
char * strncpy (char * target, char const * source, size_t n);

char * strcat  (char * target, char const * source);
char * strncat (char * target, char const * source, size_t n);

size_t strspn  (char const * string, char const * accepted);
size_t strcspn (char const * string, char const * rejected);
```

Les prototypes des fonctions en mem...() sur les blocs mémoire :

```
int    memcmp (void const * block1, void const * block2, size_t n);

void * memchr (void const * block, int byte, size_t n);
void * memset (void       * block, int byte, size_t n);

void * memcpy (void * target, void const * source, size_t n);
void * memmove(void * target, void const * source, size_t n);
```

La fonction `strlen()` de `<string.h>` (1/1)

```
size_t strlen (char const * string);
```

`strlen()` retourne le nombre de bytes dans la chaîne `string`, à l'exclusion de son zéro terminal `'\0'`.

Remarque : Comme en C, les tableaux sont indexés à partir de 0, le résultat est aussi de l'index du zéro terminal `'\0'` de la chaîne.

On peut tester le comportement de `strcmp()` comme suit :

```
void StringTest_strlen (void)
{
  assert (strlen ("banana") == 6);
  assert (strlen ("b") == 1);
  assert (strlen ("") == 0);

  assert (strlen ((char[]) { 'b', 'a', 'n', 'a', 'n', 'a', '\0' }) == 6);
  assert (strlen ((char[]) { 'b', '\0' }) == 1);
  assert (strlen ((char[]) { '\0' }) == 0);

  char text[]= "banana";
  size_t length= strlen (text);
  assert (text [length] == '\0');
}
```

La fonction `strcmp()` de `<string.h>` (1/2)

```
int strcmp (char const * string1, char const * string2);
```

`strcmp()` compare deux chaînes pour l'ordre lexicographique.
L'entier retourné est :

- `< 0` ssi `string1` précède `string2`,
- `== 0` ssi `string1` égale `string2`,
- `> 0` ssi `string1` suit `string2`.

Pour tester que `string1` et `string2` ont la même valeur :

- on doit écrire `strcmp (string1, string2) == 0`,
- et non `string1 == string2` qui teste l'égalité des adresses.

De même, pour tester que les valeurs sont différentes :

- on doit écrire `strcmp (string1, string2) != 0`,
- et non `string1 != string2` qui teste l'inégalité des adresses.

La fonction `strcmp()` de `<string.h>` (2/2)

On peut tester le comportement de `strcmp()` comme suit.
Pour l'inégalité des chaînes :

```
void StringTest_strcmp (void)
{
  // with a distinct character after the longest common prefix
  assert (strcmp ("banana", "barber") < 0);
  assert (strcmp ("barber", "banana") > 0);

  // with one string being the proper prefix of the other
  assert (strcmp ("bank", "banker") < 0);
  assert (strcmp ("banker", "bank") > 0);
  ...
```

Pour l'égalité des chaînes :

```
  // with the same object (at the same address) and hence the same value
  char text1[]= "same text";
  assert (text1 == text1);
  assert (strcmp (text1, text1) == 0);

  // with two distinct objects (at different addresses) of same value
  char text2[]= "same text";
  assert (text1 != text2);
  assert (strcmp (text1, text2) == 0);
}
```

La fonction `strncmp()` de `<string.h>` (1/2)

```
int strncmp (char const * string1, char const * string2, size_t n);
```

`strncmp()` est analogue à la fonction `strcmp()`,
mais elle examine au plus les `n` 1ers caractères des chaînes.

Si `n > min (strlen (string1), strlen (string2))`, alors
la fonction est équivalente à `strcmp (string1, string2)`.
Sinon, l'entier retourné est :

- `< 0` ssi `string1[0..n-1]` précède `string2[0..n-1]`,
- `== 0` ssi `string1[0..n-1]` égale `string2[0..n-1]`,
- `> 0` ssi `string1[0..n-1]` suit `string2[0..n-1]`.

Remarque : `[0..n-1]` n'est qu'une convention de notation ici,
pour désigner la séquence de caractères aux indices de `0` à `n-1`.
Ce n'est pas une syntaxe valide en C.

La fonction `strncmp()` de `<string.h>` (2/2)

On peut tester le comportement de `strncmp()` comme suit.
Pour l'égalité des chaînes sur les `n` 1ers caractères :

```
void StringTest_strncmp (void)
{
  assert (strncmp ("banana", "barber", 2) == 0); // "ba" equals "ba"
  assert (strncmp ("barber", "banana", 2) == 0); // "ba" equals "ba"

  assert (strncmp ("bank", "banker", 4) == 0); // "bank" equals "bank"
  assert (strncmp ("banker", "bank", 4) == 0); // "bank" equals "bank"

  assert (strncmp ("milk", "cow", 0) == 0); // "" equals ""
  ...
```

Pour l'inégalité des chaînes des chaînes sur les `n` 1ers caractères :

```
  assert (strncmp ("banana", "barber", 3) < 0); // "ban" preceeds "bar"
  assert (strncmp ("barber", "banana", 3) > 0); // "bar" follows  "ban"

  assert (strncmp ("bank", "banker", 5) < 0); // '\0' preceeds 'e'
  assert (strncmp ("banker", "bank", 5) > 0); //  'e' follows '\0'
}
```

La fonction `memcmp()` de `<string.h>` (1/2)

```
int memcmp (void const * block1, void const * block2, size_t n);
```

La fonction `memcmp()` est analogue à `strncmp()`, mais comme les blocs comparés ne sont pas nécessairement des chaînes, la rencontre d'un `'\0'` ne stoppe pas la comparaison.

On peut voir la différence entre les deux avec le test suivant :

```
void StringTest_memcmp (void)
{
  assert (strncmp("abc\0\0\0def", "abc\0\0\0xyz", 3) == 0);
  assert (strncmp("abc\0\0\0def", "abc\0\0\0xyz", 9) == 0);

  assert (memcmp ("abc\0\0\0def", "abc\0\0\0xyz", 3) == 0);
  assert (memcmp ("abc\0\0\0def", "abc\0\0\0xyz", 9) <  0);

  assert (memcmp ("abc\0\0\0def", "abc\0\0\0def", 9) == 0);
}
```

La fonction strchr() de <string.h>

```
char * strchr (char const * string, int character);
```

strchr() recherche la 1ère occurrence de character dans string.
Si aucune occurrence n'est trouvée, NULL est retourné.
Si une occurrence est trouvée, son adresse occ est retournée.

Remarque 1 : On peut déduire son index occ_index dans string par soustraction de pointeurs : occ_index= occ - string.
Remarque 2 : character peut être le zéro terminal '\0'.

On peut tester le comportement de strchr() comme suit :

```
void StringTest_strchr (void)
{
  char text[]= "banana split";
  assert (strchr (text,  'z') == NULL);
  assert (strchr (text, '\0') == text + strlen (text));
  char * occ1= strchr (text    , 'a'); assert (occ1 == text + 1);
  char * occ2= strchr (occ1 + 1, 'a'); assert (occ2 == text + 3);
  char * occ3= strchr (occ2 + 1, 'a'); assert (occ3 == text + 5);
  char * occ4= strchr (occ3 + 1, 'a'); assert (occ4 == NULL);
}
```

La fonction strrchr() de <string.h>

```
char * strrchr (char const * string, int character);
```

strrchr() est analogue à la fonction strchr()
mais recherche la dernière occurrence de character dans string.
Si aucune occurrence n'est trouvée, NULL est retourné.
Si une occurrence est trouvée, son adresse occ est retournée.

Remarque 1 : On peut déduire son index occ_index dans string par soustraction de pointeurs : occ_index= occ - string.
Remarque 2 : character peut être le zéro terminal '\0'.

On peut tester le comportement de strrchr() comme suit :

```
void StringTest_strrchr (void)
{
  char text[]= "banana split";
  char * occ= strrchr (text, 'a');
  assert (occ == text + 5);
  assert (strrchr (text,  'z') == NULL);
  assert (strrchr (text, '\0') == text + strlen (text));
}
```

La fonction `memchr()` de `<string.h>`

```
void * memchr (void const * block, int byte, size_t n);
```

La fonction `memchr()` recherche la première occurrence de `byte` (interprété comme un `unsigned char`) dans le bloc `block`.
Si une occurrence trouvée dans les `n` premiers bytes,
son adresse est retournée, sinon la valeur `NULL` est retournée.

Comme le bloc n'est pas nécessairement une chaîne,
la rencontre d'un `'\0'` ne stoppe pas la recherche,
contrairement à la fonction `strchr()`.

On peut voir la différence entre les deux comme suit :

```
void StringTest_memchr (void)
{
  char buffer[]= "abc\0def";
  assert (strchr (buffer, 'e')    == NULL);
  assert (memchr (buffer, 'e', 3) == NULL);
  assert (memchr (buffer, 'e', 6) == buffer+5);
}
```

La fonction strpbrk() de <string.h>

```
char * strpbrk (char const * string, char const * accept);
```

strpbrk() recherche la 1ère occurrrence d'un caractère
qui soit aussi élément de l'ensemble accept.
Si aucune occurrence n'est trouvée, NULL est retourné.
Si une occurrence est trouvée, son adresse occ est retournée.

On peut tester le comportement de strpbrk() comme suit :

```
void StringTest_strpbrk (void)
{
  char text[]= "mic mac\n";
  char accept[]= " \n";
  char * occ1= strpbrk (text     , accept); assert (occ1 == text+3);
  char * occ2= strpbrk (occ1 + 1, accept); assert (occ2 == text+7);
  char * occ3= strpbrk (occ2 + 1, accept); assert (occ3 == NULL);
}
```

La fonction `strstr()` de `<string.h>`

```
char * strstr (char const * string, char const * substring);
```

`strstr()` recherche la 1ère occurrence de la chaîne `substring`.
Si aucune occurrence n'est trouvée, `NULL` est retourné.
Si une occurrence est trouvée, son adresse `occ` est retournée.

Remarque : Si `substring` est la chaîne vide `""`, alors
elle est trouvée en début de chaîne, et `occ == string`.

On peut tester le comportement de `strstr()` comme suit :

```
void StringTest_strstr (void)
{
  char text[]= "banana split";
  char subtext[]= "ana";
  char * occ1= strstr (text    , subtext); assert (occ1 == text + 1);
  char * occ2= strstr (occ1 + 1, subtext); assert (occ2 == text + 3);
  char * occ3= strstr (occ2 + 1, subtext); assert (occ3 == NULL);
  assert (strstr (text, "") == text);
}
```

La fonction `strcpy()` de `<string.h>`

```
char * strcpy (char * target, char const * source);
```

`strcpy()` copie la chaîne `source` vers le tableau `target`.
Celui-ci doit avoir au moins la taille `strlen(source)+1`,
pour stocker la chaîne `source`, zéro terminal `'\0'` inclus.
L'adresse retournée, peu utile, est `target`.

On peut tester le comportement de `strcpy()` comme suit :

```
void StringTest_strcpy (void)
{
  char source[]= "banana split";
  size_t length= strlen(source);
  char target[length+1];
  char * result= strcpy (target, source);
  assert (strcmp (source, target) == 0);
  assert (result == target);
}
```

Remarque : Hormis pour la valeur de retour,
`strcpy (target, source)` est équivalent à
`sprintf (target, "%s", source)`.

La fonction strncpy() de <string.h>

```
char * strncpy (char * target, char const * source, size_t n);
```

La fonction strncpy() est analogue à strcpy(), mais la copie s'arrête soit après la copie du '\0', soit après n caractères. L'espace disponible restant est ensuite rempli avec des '\0'.

Remarque : strcpy(target, source) est équivalent à strncpy(target, source, 1+strlen (source)).

Attention : Si n < 1+strlen(source), le '\0' n'est pas copié.

On peut tester le comportement de strncpy() comme suit :

```
void StringTest_strncpy (void)
{
  char text[]= "abc";
  char copy[]= "xxxxxx";
  char * result1= strncpy (copy, text, 3);
  assert (result1 == copy && memcmp (copy, "abcxxx", 6) == 0);
  char * result2= strncpy (copy, text, 6);
  assert (result2 == copy && memcmp (copy, "abc\0\0\0", 6) == 0);
}
```

Les fonctions memcpy() et memmove() de <string.h>

```
void * memcpy  (void * target, const void * source, size_t n);
void * memmove (void * target, const void * source, size_t n);
```

La fonction memcpy() copie n bytes de source vers target.
Comme le bloc copié n'est pas nécessairement une chaîne,
la rencontre d'un '\0' ne stoppe pas la copie comme strncpy().
La valeur retournée est juste le pointeur target.

La fonction memmove() fait la même chose que memcpy(),
mais avec memmove(), les blocs peuvent se chevaucher :

```
void StringTest_memcpy_memmove (void)
{
  char buffer1[]= "abc\0def\0ghi";
  char * result1= memcpy (buffer1, buffer1 + 6, 3); // no overlap
  assert (result1 == buffer1 && memcmp (buffer1, "f\0g\0def\0ghi", 11) == 0);

  char buffer2[]= "abc\0def\0ghi";
  char * result2= memmove (buffer2, buffer2 + 4, 7); // overlap
  assert (result2 == buffer2 && memcmp (buffer2, "def\0ghi\0ghi", 11) == 0);

  char buffer3[]= "abc\0def\0ghi";
  char * result3= memmove (buffer3 + 4, buffer3, 7); // overlap
  assert (result3 == buffer3 + 4 && memcmp (buffer3, "abc\0abc\0def", 11) == 0);
}
```

La fonction `memset()` de `<string.h>`

```
void * memset (void * block, int byte, size_t n);
```

La fonction `memset()` remplit les bytes du bloc mémoire `block` avec la valeur de `byte`, interprété comme un `unsigned char`.
La valeur retournée est juste le pointeur `block`.

On peut tester le comportement de `memset()` comme suit :

```
void StringTest_memset (void)
{
  char buffer[]= "xxxxxx";
  char * result= memset (buffer, 'a', 3);
  assert (result == buffer && memcmp (buffer, "aaaxxx", 6) == 0);
}
```

Les fonctions `strcat()` et `strncat()` de `<string.h>`

```
char * strcat  (char * target, char const * source);
char * strncat (char * target, char const * source, size_t n);
```

La fonction `strcat()` concatène la chaîne `source`
à la suite de la chaîne `target`.
La valeur retournée est juste le pointeur `target`.

La fonction `strncat()` fait la même chose,
mais ne copie qu'au plus `n` caractères de `source`.
Un `'\0'` est rajouté à la fin si `n < 1+strlen(source)`.

On peut tester de `strcat()` et `strncat()` comme suit :

```
void StringTest_strcat_strncat (void)
{
  char buffer[100];
  strcpy (buffer, "banana" );
  strcat (buffer, " split" );
  assert (strcmp (buffer, "banana split") == 0);
  strncat(buffer, "...xxx", 3);
  assert (strcmp (buffer, "banana split...") == 0);
}
```

Les fonctions `strspn()` et `strcspn()` de `<string.h>`

```
size_t strspn  (char const * string, char const * accepted);
size_t strcspn (char const * string, char const * rejected);
```

`strspn()` retourne la longueur du préfixe de `string` (ou *span*) constitué uniquement de caractères dans l'ensemble `accepted`.

`strcspn()` retourne la longueur du préfixe de `string` (ou *co-span*) constitué uniquement des caractères hors de l'ensemble `rejected`.

On peut tester leur comportement comme suit :

```
void StringTest_strspn_strcspn (void)
{
  char text[]= "  Regis \n ";
  char blanks[]= " \n";

  size_t n1= strspn  (text      , blanks); assert (n1 == 2);
  size_t n2= strcspn (text+n1   , blanks); assert (n2 == 5);
  size_t n3= strspn  (text+n1+n2, blanks); assert (n3 == 3);

  assert (text [n1+n2+n3] == '\0');
}
```

La fonction strcoll() de <string.h>

La fonction strcoll() compare deux chaînes,
mais en étant sensible à la locale LC_COLLATE.
Pour la locale "C", strcoll() se comporte comme strcmp().
On peut tester les différences de comportement comme suit :

```
void StringTest_strcoll_localeC (void)
{
  char * old_collate= setlocale (LC_COLLATE, NULL);
  setlocale (LC_COLLATE, "C");
  assert (strcmp ("e", "f") < 0  &&  strcmp ("f", "\xE9") < 0);     // e < f < é
  assert (strcoll("e", "f") < 0  &&  strcoll("f", "\xE9") < 0);     // e < f < é

  setlocale (LC_COLLATE, "fr_FR.ISO-88591");
  assert (strcmp ("e",    "f") < 0  &&  strcmp ("f", "\xE9") < 0); // e < f < é
  assert (strcoll("e", "\xE9") < 0  &&  strcoll("\xE9", "f") < 0); // e < é < f

  setlocale (LC_COLLATE, "fr_FR.UTF-8");
  assert (strcmp ("e", "f") < 0  &&  strcmp ("f", "é") < 0);        // e < f < é
  assert (strcoll("e", "é") < 0  &&  strcoll("é", "f") < 0);        // e < é < f

  assert (strcmp ("e", "f"       ) < 0  &&  strcmp ("f", "\xC3\xA9") < 0);
  assert (strcoll("e", "\xC3\xA9") < 0  &&  strcoll("\xC3\xA9", "f") < 0);
  setlocale (LC_COLLATE, old_collate);
}
```

Remarque 1 : en ISO-88591 (aka Latin-1), é est encodé par \xE9.
Remarque 2 : en UTF8, é est encodé par le doublet \xC3\xA9.