



Nom :
Prénom :
No. groupe :
No. carte :

Programmation et structures de données en C– LU2IN018

Partiel du 18 novembre 2021

1h30

Aucun document n'est autorisé. Le memento qui a été distribué est reproduit à la fin de cet énoncé.

Les calculatrices, baladeurs et autres appareils électroniques sont interdits. Les téléphones mobiles doivent être éteints et rangés dans les sacs.

Toutes les questions sont indépendantes. Pour les questions à choix multiples à 1 point, vous obtenez 1 point si vous avez coché toutes les cases correspondant à des réponses justes et seulement celles-ci. Pour les questions à 2 points ou plus acceptant plusieurs réponses, vous perdez 1 point par réponse erronée (case juste non cochée ou case fausse cochée). La note minimale à une question est 0. Le barème sur 25 points (15 questions) n'a qu'une valeur indicative.

ATTENTION : lisez le sujet dans son intégralité avant de commencer. Certaines questions sont à réponse libre (lecture ou écriture de code). Elles peuvent donc nécessiter plus de temps de réflexion que les questions à choix multiples.

Il ne vous est pas demandé de vérifier qu'un `malloc` a bien alloué la mémoire. De même il n'est pas demandé de vérifier qu'un `fopen` a ouvert correctement le fichier demandé.

Question 1 (1 point)

Soit les structures suivantes définies dans un fichier catalogue.h :

```

typedef struct _article1 {
    int numero;
    char libelle[50];
    char *commentaire;
} Article1;

typedef struct _article2 {
    int numero;
    char libelle[50];
    char *commentaire;
    struct _article2 *suiv;
} Article2;
  
```

```

typedef struct _article3 {
    char libelle[50];
    char *commentaire;
} Article3;

typedef struct _article4 {
    char libelle[50];
    char commentaire[301];
} Article4;

```

On souhaite faire un catalogue d'articles de papeterie comportant environ une centaine d'articles. Ces articles ont tous un numéro unique associé : le premier article du catalogue est l'article 0, le second est l'article 1, etc. Le commentaire accompagnant un article peut atteindre 300 caractères, mais sera en pratique bien plus court la plupart du temps. Le catalogue est stable, il n'y a presque jamais d'articles supprimés du catalogue et peu sont ajoutés a posteriori.

Dans cette optique, pour créer le catalogue, on peut utiliser soit une liste chaînée dont chaque maillon décrit un article, soit un tableau dont chaque case décrit un article. Cochez la solution la plus appropriée pour le tableau (celle qui prendra le moins de place en mémoire) :

- Article1 tab1[100];
- Article2 tab2[100];
- Article3 tab3[100];
- Article4 tab2[100];

Question 2 (2 points)

Si l'on ne sait pas si le catalogue a une structure stable, c'est-à-dire qu'il peut être possible d'avoir à rajouter ou supprimer souvent des articles de ce catalogue, quelle va être la représentation que l'on va en choisir en mémoire, avec quelle structure parmi les quatre ci-dessus et pourquoi ? Remarque : dans ce cas, on ne fera pas l'hypothèse que les numéros des articles doivent obligatoirement se suivre.

Solution: On va choisir la représentation sous forme de liste chaînée avec des maillons de type Article2, car cette représentation est plus souple que celle réalisée avec un tableau. S'il faut supprimer un article du catalogue, il suffit de supprimer le maillon correspondant et de récupérer l'espace mémoire qu'il occupe, ajouter un article revient à ajouter un maillon, par exemple en tête de liste, ce qui n'est pas couteux en terme de temps ou d'espace mémoire.

Lorsque l'on retire un article d'une case du tableau, on ne peut pas récupérer intégralement la place occupée (on peut tout de même libérer l'espace mémoire que l'on a alloué dynamiquement pour remplir le champ commentaire correspondant). Il faut ensuite décaler les éléments pour enlever la case "vide". L'ajout d'un article lorsque le tableau est plein n'est possible qu'avec une reallocation de taille plus grande, ce qui peut nécessiter de recopier tout le tableau, ce qui peut prendre du temps.

1pt pour la bonne réponse (liste chaînée et structure Article2), 1pt pour la justification (ajout et suppression plus efficace).

Question 3 (2 points)

On déclare un tableau de la façon suivante (sans se préoccuper de savoir si c'est la meilleure façon de faire ou pas...) :

Article1 tab1[100];

Aucune autre instruction n'est exécutée avant les instructions indiquées ci-après.

Cochez la ou les réponses correctes :

- L'ensemble d'instructions suivant est correct :

```
char stylo[]="Joli_stylo_à_plume_doré_à_l'or_fin";
strcpy(tab1[6].libelle, stylo);
```

L'instruction suivante est correcte :

```
strcpy(tab1[6].commentaire, "C'est_un_stylo_très_précieux,_de_
collection_créé_par_Vaterman.");
```

- Une fois les différents éléments initialisés, l'espace mémoire total occupé par l'ensemble des informations relatives à un article n'est pas toujours le même.
- Si l'on n'a plus besoin de l'article figurant dans une case donnée, la case doit être libérée.

Question 4 (2 points)

Ecrire la fonction de prototype :

```
Article2 *creer_article2(int num, char *libel, char *comment);
```

qui crée un nouvel article dans une liste chaînée de maillons de type Article2.

Solution:

```
Article2 creer_article2(int num, char *libel, char *comment) {
    Article2 *art=(Article2*) malloc(sizeof(Article2));
    art->numero=num;
    strcpy(art->libelle,libel);
    art->commentaire=strdup(comment);
    art->suiv=NULL;
    return art;
}
```

Question 5 (2 points)

Ecrire la fonction de prototype :

```
void liberer(Article2 *liste);
```

qui libère l'espace mémoire occupé par une liste chaînée dont les maillons sont de type Article2.

Solution:

```
void liberer(Article2 *liste) {
    Article2 *tmp;
    while (liste) {
        free(liste->commentaire);
        tmp=liste->suivant;
        free(liste);
        liste=tmp;
    }
}
```

Question 6 (3 points)

Soit la fonction de prototype :

```
void g(char a, int **b, char *c, char **d, int *e);
```

Lorsqu'il est question de modification d'une variable dans g, il s'agit de modification qui puisse perdurer après l'appel à g.

Cochez la ou les réponse(s) correct(e)s :

- Le paramètre a permet de transmettre un caractère dont on peut modifier la valeur dans g.
- Le paramètre c peut représenter un caractère transmis par adresse, sa valeur peut donc être modifiée dans g.
- c peut être l'adresse d'un tableau de caractères dont le contenu ne peut pas être modifié dans g.
- d peut désigner un pointeur vers l'adresse d'un tableau de caractères et on peut s'en servir dans g pour faire une modification telle que : (*d)[5]='7' ;
- d peut désigner un pointeur vers l'adresse d'un tableau de caractères et on peut s'en servir dans g pour faire une modification telle que : (**d)[5]='7' ;
- Les instructions suivantes sont correctes :

```
int compte=7;
*b=compte;
```

- Il est possible de modifier dans g la valeur de l'entier pointé par e.

Question 7 (2 points)

On définit le programme suivant dans le fichier catalogue.c :

```

1      #include <stdio.h>
2      #include <stdlib.h>
3      #include <string.h>
4      /* Instruction0 : inclusion du fichier catalogue.h */
5      #define TAILLE 100
6
7      void insertion_catalogue(Article3 *catalogue, int num, char *
8          libel,
9          char *comment) {
10         /* libel et comment sont susceptibles d'être libérés
11            juste après cet appel */
12         Article3 art;
13         /* Instruction 1: à trouver... */
14         /* Instruction 2: à trouver... */
15         catalogue[num]=art;
16     }
17     void affiche_article(Article3 tab[], int num) {
18         if(tab[num].commentaire!=NULL) {
19             /* une case du tableau dont le commentaire
20                est NULL sera supposée être vide */
21             printf("Article_no_%d\n",num);
22             printf("%s\n",tab[num].libelle);
23             printf("%s\n",tab[num].commentaire);
24         }
25     }
26     int main(void) {
27         int i;
28         Article3 catalogue[TAILLE];
29         insertion_catalogue(catalogue,45,"gomme_plastique",
30             une_gomme_superbe,_design,_bleue_et_blanche_avec_
31             logo_Droit_au_But");

```

```

28         insertion_catalogue(catalogue, 255, "règle_30cm_"
29             plastique", "une_règle_incassable,_fabriquée_en_"
30                 France");
31         affiche_article(catalogue, 45);
32         affiche_article(catalogue, 255);
33         for (i=0;i<TAILLE;i++) {
34             free(catalogue[i].commentaire);
35         }
36         return 0;
37     }

```

Cochez la ou les réponses correctes :

- Instruction0 peut être remplacée par une ligne vide
- Instruction0 doit être remplacée par #include <catalogue.h>
- Instruction0 doit être remplacée par #include "catalogue.h"

- Instruction1 doit être remplacée par : art.libelle=libel;
- Instruction1 doit être remplacée par : strcpy(art.libelle,libel);
- Instruction1 doit être remplacée par art.libelle=strdup(libel);

- Instruction2 doit être remplacée par : art.commentaire=comment;
- Instruction2 doit être remplacée par : strcpy(art.commentaire,comment);
- Instruction2 doit être remplacée par : art.commentaire=strdup(comment);

Question 8 (1 point)

On compile, puis exécute le programme à l'aide des lignes de commandes suivantes :

```

bash$ gcc -g -Wall -o catalogue catalogue.c
bash$ ./catalogue
Segmentation fault (core dumped).

```

Cochez la ou les réponses correctes :

- Segmentation fault est un warning sans conséquence sur l'exécution du programme.
- gcc peut donner des indications pour réparer cette erreur, si erreur il y a.
- C'est éventuellement une erreur syntaxique détectée par le compilateur.
- Il s'agit d'une erreur grave à l'exécution.
- ddd peut permettre de trouver l'erreur commise dans ce programme s'il y en a une.

Question 9 (2 points)

On décide d'utiliser valgrind sur ce programme à l'aide de la commande valgrind ./catalogue.

On obtient le résultat suivant :

```

==744016== Memcheck, a memory error detector
==744016== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==744016== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==744016== Command: ./catalogue
==744016==
==744016== Invalid write of size 8
==744016==   at 0x109252: insertion_catalogue (catalogue.c:13)
==744016==   by 0x109394: main (catalogue.c:28)
==744016== Address 0x1fff002370 is not stack'd, malloc'd or (recently) free'd

```

```

==744016==
==744016==
==744016== Process terminating with default action of signal 11 (SIGSEGV)
==744016== Access not within mapped region at address 0x1FFF002370
==744016==   at 0x109252: insertion_catalogue (catalogue.c:13)
==744016==   by 0x109394: main (catalogue.c:28)
==744016== If you believe this happened as a result of a stack
==744016== overflow in your program's main thread (unlikely but
==744016== possible), you can try to increase the size of the
==744016== main thread stack using the --main-stacksize= flag.
==744016== The main thread stack size used in this run was 8388608.
==744016==
==744016== HEAP SUMMARY:
==744016==     in use at exit: 111 bytes in 2 blocks
==744016==   total heap usage: 2 allocs, 0 frees, 111 bytes allocated
==744016==
==744016== LEAK SUMMARY:
==744016==   definitely lost: 0 bytes in 0 blocks
==744016==   indirectly lost: 0 bytes in 0 blocks
==744016==   possibly lost: 0 bytes in 0 blocks
==744016==   still reachable: 111 bytes in 2 blocks
==744016==       suppressed: 0 bytes in 0 blocks
==744016== Rerun with --leak-check=full to see details of leaked memory
==744016==
==744016== For lists of detected and suppressed errors, rerun with: -s
==744016== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Erreur de segmentation (core dumped)

```

Expliquez cette erreur d'après la trace de valgrind (on fera l'hypothèse que ce n'est pas lié aux instructions insérées) en indiquant les lignes incriminées et la cause du problème.

Solution: Valgrind renseigne sur le type d'erreur qui est une "segmentation fault", un problème de mémoire, d'écrasement illicite d'une portion de la mémoire et l'erreur est signalée sur l'instruction : catalogue[num]=art;

qui est à la ligne 13 du programme, elle-même appelée par l'instruction qui est à la ligne 28.

Il vient naturellement à l'esprit qu'il s'agit d'une écriture sur une portion de mémoire qui n'est pas allouée au tableau catalogue. Il s'agit d'une écriture hors des bornes du tableau . La taille du tableau catalogue est donné par TAILLE dont la valeur est précisée dans un define et égale à 100, alors que dans le main, on a l'instruction : affiche_article(catalogue, 255) ;

Question 10 (1 point)

Comment faut-il modifier le programme pour supprimer l'erreur ?

Solution: Le mieux est de tester le programme avec un catalogue de TAILLE 2 et les numéros 0 et 1 pour les deux articles ajoutés. Il est possible également d'agrandir la taille du tableau à 300, par exemple, en modifiant :

```
#define TAILLE 100
```

```
en
```

```
#define TAILLE 300
```

ou de choisir pour l'article que l'on voulait placer en case 255, une case de numéro compris entre 0 et 99. Cette solution résoudra le problème de segmentation, mais elle est moins satisfaisante que la première, sachant que cela risque de provoquer une erreur de segmentation lors de la boucle de libération des commentaires, si jamais un de ces champs n'a pas la valeur NULL après sa création (ce n'est pas systématique, mais c'est possible).

Barème : 1pt pour la première solution, 0.5 pour la 2eme.

Question 11 (1 point)

On a fait un catalogue d'articles décrits chacun par un numéro, un libellé et un commentaire, catalogue stocké dans une liste chaînée de maillons de structure Article2, on a sauvegardé ce catalogue sur un premier fichier catalogue_sans_prix.txt en stockant les informations relatives à un article sur 3 lignes consécutives contenant respectivement le numéro de l'article, son libellé et un commentaire relatif à l'article. Après une étude de la concurrence, on est maintenant à même de fixer un prix de vente pour les articles du catalogue. On écrit donc un programme qui lit sur le fichier catalogue_sans_prix.txt les informations relatives à chaque article, les affiche à l'écran, demande à l'utilisateur de saisir le prix (float) et enfin, écrit sur un nouveau fichier catalogue_avec_prix.txt les caractéristiques précédentes de l'article auxquelles s'ajoute maintenant son prix, sous le même format que précédemment, c'est-à-dire une information par ligne. Le programme suivant réalise cette tâche pour un seul article :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    int numero;
    float prix;
    char libelle[100];
    char commentaire[100];
    FILE *fl=fopen("catalogue_sans_prix.txt","r");
    FILE *fe=fopen("catalogue_avec_prix.txt","w");
    /* Instr1 : lecture du numéro de l'article */
    /* Instr2 : lecture du libellé de l'article */
    /* Instr3 : lecture du commentaire au sujet de l'article
    */
    printf("\nVeuillez_saisir_le_prix_(en_euros)_de_l'article
    _suivant:\n");
    printf("Numéro=%d\n",numero);
    printf("Libellé=%s\n",libelle);
    printf("Commentaire=%s\n",commentaire);
    /*Instr4 : saisie du prix de l'article */
    /*Instr5 : écriture de l'article */
    return 0;
}
```

NB : On ne se préoccupe pas des retours à la ligne et les noms d'articles ne contiennent pas d'espace. En supposant que les instructions Instr1,...,Instr5 soient complétées convenablement, ce programme est-il complet ? Si non, que faut-il ajouter et où ?

Solution: Il est nécessaire de fermer, à la fin du programme, le fichier ouvert en lecture catalogue_sans_prix.txt et le fichier ouvert en écriture catalogue_avec_prix.txt. On ajoute donc, juste avant l'instruction return 0 ; du main :

```
fclose(f1);
fclose(fe);
```

Question 12 (2 points)

Cochez la ou les instructions qui vous semblent correctes pour Instr3, Instr4 et Instr5 :

- Instr3 : fscanf(fe, " %s", commentaire);
- Instr3 : fscanf(f1, " %s", commentaire);
- Instr3 : fgets(commentaire, 100, f1);
- Instr3 : fread(commentaire, 100, 100, f1);

- Instr4 peut s'écrire :getc(&prix);
- Instr4 peut s'écrire : scanf(" %f", prix);
- Instr4 peut s'écrire : scanf(" %f", &prix);
- Instr4 peut s'écrire : printf(" %f", &prix);

- Instr5 peut s'écrire : fprintf(fe, "%d\n%s\n%s\n%f\n", numero, libelle, commentaire, prix);
- Instr5 peut s'écrire : fprintf(f1, "%d\n%s\n%s\n%f\n", numero, libelle, commentaire, prix);
- Instr5 peut s'écrire : fwrite(fe, 100, 100, numero, libelle, commentaire, prix);
- Instr5 peut s'écrire : fwrite(f1, 100, 100, numero, libelle, commentaire, prix);

Question 13 (1 point)

Cochez la ou les réponses correctes :

- Dans la trace d'une compilation, on peut systématiquement ignorer les warnings, ceux-ci n'ayant aucune importance.
- Toute utilisation inappropriée de la mémoire donne lieu à un message de type Segmentation fault.
- Une fuite mémoire résulte de l'absence d'une désallocation correcte de la mémoire qui a été précédemment allouée.
- Valgrind ne permet pas de détecter des fuites mémoire.
- ddd permet de déboguer un programme au travers d'une interface graphique.

Question 14 (2 points)

Cochez la ou les réponses correctes :

- L'outil make avec le fichier Makefile associé permet de compiler un programme en explicitant les dépendances de fichiers les uns par rapport aux autres et donc de ne compiler que ce qui est nécessaire.
- Grâce à make, il n'y a rien à faire pour obtenir la nouvelle version d'un exécutable après modification d'un de ses fichiers source.
- Les fichiers headers de suffixe .h permettent d'utiliser les fonctions dans un fichier qui ne contient pas leur déclaration.
- Le pré-processeur intervient avant le compilateur et son rôle est de pré-traiter un programme en le complétant grâce aux directives précédées par # comme #define ou #include.

- Un fichier header, outre les prototypes de certaines fonctions, peut contenir la définition de structures C.
- `#define TAILLE = 100` est un exemple de directive valide pour le pré-processeur.

Question 15 (1 point)

Supposons que l'on ait un programme divisé en 3 fichiers : `prog_main.c`, `fich_lecture.c` et `fich_ecriture.c`. On a aussi créé deux fichiers headers `fich_lecture.h` et `fich_ecriture.h`, qui contiennent respectivement les prototypes des fonctions de `fich_lecture.c` et de `fich_ecriture.c`. On veut créer un exécutable nommé `prog`. On peut faire cela grâce à la séquence de commandes suivantes :

- `gcc -Wall -o prog prog_main.c fich_lecture.c fich_ecriture.c`
- `gcc -c -Wall -o prog_main.o prog_main.c`
`gcc -c -Wall -o fich_lecture.o fich_lecture.c`
`gcc -c -Wall -o fich_ecriture.o fich_ecriture.c`
`gcc -Wall -o prog prog_main.o fich_lecture.o fich_ecriture.o`
- `gcc -Wall -h fich_lecture.h fich_ecriture.h`
`gcc -Wall -o prog prog_main.o fich_lecture.o fich_ecriture.o`

gcc -c -Wall -o prog_main.c prog_main.o
gcc -c -Wall -o fich_lecture.c fich_lecture.o
gcc -c -Wall -o fich_ecriture.c fich_ecriture.o
gcc -Wall -o prog_main.o fich_lecture.o fich_ecriture.o prog

Mémento de l'UE LU2IN018

Ce document a pour vocation de présenter un bref descriptif des fonctions offertes par les bibliothèques standards et qui sont susceptibles d'être utilisées dans l'UE.

Entrées - sorties

Prototypes disponibles dans `stdio.h`.

Entrées, sorties formatées

```
int printf(const char *format, ...);
```

La fonction `printf` écrit sur la sortie standard (par défaut le terminal), un message correspondant au texte spécifié dans la chaîne `format`. Le texte affiché peut contenir des portions *variables* spécifiées par des codes de conversion précédés par le caractère `%`. Les arguments suivant `format` doivent correspondre (ordre et type) aux codes de conversion présents. Voici quelques codes de conversion courants :

- `%d` : entier
- `%c` : caractère
- `%s` : chaîne de caractères
- `%f` : nombre réel

`printf` renvoie le nombre de caractères imprimés et la chaîne de format peut contenir des codes de contrôle permettant le formatage comme `\n` pour forcer le passage à la ligne et `\t` pour insérer une tabulation.

```
int scanf (const char *format, ...);
int sscanf(const char *entree, const char *format, ...);
```

Les fonctions `scanf` et `sscanf` permettent de saisir et analyser un texte saisi sur l'entrée standard, par défaut le clavier (`scanf`) ou depuis une chaîne de caractères passée en argument (`sscanf`). Le texte devra respecter le `format` spécifié et les arguments suivants doivent correspondre à des pointeurs sur des variables de type appropriés. Les codes de conversion sont identiques à ceux de `printf`.

Entrées, sorties caractères

```
int getchar(void);
```

Lit un caractère dans le flux d'entrée standard (par défaut le clavier). La fonction retourne le code `EOF` en cas d'erreur, le caractère lu dans le cas contraire.

```
int putchar(int c);
```

Affiche un caractère dans le flux de sortie standard (par défaut le terminal). La fonction retourne le code `EOF` en cas d'erreur, le caractère lu dans le cas contraire.

```
int puts(const char *s);
```

Affiche une chaîne de caractères dans le terminal et passe à la ligne, renvoie `EOF` en cas d'erreur.

Fichiers

Prototypes disponibles dans `stdio.h`.

```
FILE *fopen (const char *path, const char *mode);
```

Ouvre un fichier dont le chemin est spécifié par la chaîne `path` et retourne un pointeur de type `FILE *` (`NULL` en cas d'échec). L'argument `mode` permet de spécifier le type d'accès à réaliser sur le fichier :

- `[r]` pour un accès en lecture,
- `[w]` pour un accès en écriture et le contenu précédent du fichier est écrasé,
- `[a]` pour un accès en écriture, le contenu du fichier est préservé et les écritures sont effectuées à la suite des contenus déjà présents.

En cas d'erreur la fonction retourne le code `NULL` sinon un pointeur vers le fichier ouvert.

```
int fclose(FILE *fp);
```

Cette fonction provoque la fermeture du fichier pointé par `fp`. En cas d'erreur la fonction retourne le code `EOF` sinon 0.

```
int fprintf(FILE *stream, const char *format, ...);
```

Identique à `printf` mais l'argument `stream` permet de spécifier le flux de sortie.

```
int fscanf(FILE *stream, const char *format, ...);
```

Identique à `scanf` mais l'argument `stream` permet de spécifier le flux d'entrée.

```
int fputc(int c, FILE *stream);
```

Identique à `putchar` mais l'argument `stream` permet de spécifier le flux de sortie.

```
int fputs(const char *s, FILE *stream);
```

Identique à `puts` mais l'argument `stream` permet de spécifier le flux de sortie et il n'y a pas d'ajout de passage à la ligne.

```
int getc(FILE *stream);
```

Identique à `getchar` mais l'argument `stream` permet de spécifier le flux d'entrée.

```
char *fgets(char *s, int size, FILE *stream);
```

Lit au plus `size-1` octets dans le flux `stream`. La lecture s'arrête dès qu'un passage à la ligne est rencontré. Les octets lus sont stockés dans `s`. La fonction retourne `s` en cas de succès et `NULL` en cas d'erreur.

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Lecture binaire de **nmemb** éléments de **size** octets dans le fichier **stream**. Les données lues sont stockées en mémoire à partir de l'adresse **ptr**. La fonction retourne le nombre d'éléments effectivement lus.

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
FILE *stream);
```

Écriture de **nmemb** éléments de **size** octets dans le fichier **stream**. Les données à écrire sont lues en mémoire à partir de l'adresse **ptr**. La fonction retourne le nombre d'éléments effectivement écrits.

Chaînes de caractères

Prototypes disponibles dans **string.h**.

Une chaîne de caractères correspond à un tableau de caractère et doit contenir un marqueur de fin \0.

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

Renvoie la longueur d'une chaîne de caractères (marqueur de fin \0 non compris).

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

```
int strncmp(const char *s1, const char *s2, size_t n);
```

Comparaison entre chaînes de caractères éventuellement limité aux **n** premiers caractères. La valeur renournée est :

- 0 si les deux chaînes sont identiques,
- négative si **s1** précède **s2** dans l'ordre lexicographique (généralisation de l'ordre alphabétique),
- positive sinon.

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

```
char *strncpy(char *dest, const char *src, size_t n);
```

Copie le contenu de la chaîne **src** dans la chaîne **dest** (marqueur de fin \0 compris). La chaîne **dest** doit avoir précédemment été allouée. La copie peut être limitée à **n** caractères et la valeur renournée correspond au pointeur de destination **dest**.

```
void *memcpy(void *dest, const void *src, size_t n);
```

Copie **n** octets à partir de l'adresse contenue dans le pointeur **src** vers l'adresse stockée dans **dest**. **dest** doit pointer vers une zone mémoire préalablement allouée et de taille suffisante. **memcpy** renvoie la valeur de **dest**.

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

Retourne le nombre de caractères de la chaîne **s** (marqueur de fin \0 non compris).

```
char * strdup(const char *s);
```

Cette fonction permet de dupliquer une chaîne de caractères, elle retourne un pointeur vers la chaîne nouvellement allouée. La nouvelle chaîne pourra être libérée avec la fonction **free**.

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

Ajoute la chaîne **src** à la suite de la chaîne **dst**. La chaîne **dest** devra avoir été allouée et être de taille suffisante. La fonction retourne **dest**.

```
char * strstr(const char *haystack, const char *needle);
```

La fonction renvoie un pointeur sur la première occurrence de la sous-chaîne **needle** rencontrée dans la chaîne **haystack**. Si la chaîne recherchée n'est pas présente, la fonction retourne NULL.

Conversion de chaînes de caractères

Prototypes disponibles dans **stdlib.h**.

```
int atoi(const char *nptr);
```

La fonction convertit le début de la chaîne pointée par **nptr** en un entier de type **int**.

```
double atof(const char *nptr);
```

Cette fonction convertit le début de la chaîne pointée par **nptr** en un **double**.

```
long int strtol(const char *nptr, char **endptr, int base);
```

Convertit le début de la chaîne **nptr** en un entier long. L'interprétation tient compte de la **base** et la variable pointée par **endptr** est affectée avec l'adresse du premier caractère invalide (au sens de la conversion).

Allocation dynamique de mémoire

Prototypes disponibles dans **stdlib.h**.

```
void * malloc(size_t size);
```

Alloue **size** octets de mémoire et retourne un pointeur générique correspondant à l'adresse du premier octet de la zone, renvoie NULL en cas d'échec.

```
void * realloc(void *ptr, size_t size);
```

Permet de modifier la taille d'une zone de mémoire allouée dynamiquement. **ptr** doit correspondre à l'adresse du premier octet de la zone précédemment allouée par **malloc** ou **realloc**. **size** correspond à la taille en octet de la nouvelle zone allouée. **realloc** garantie que la nouvelle zone contiendra les données présentes dans la zone initiale.

```
void free(void *ptr);
```

Libère une zone mémoire allouée dynamiquement. **ptr** doit correspondre à l'adresse du premier octet de la zone précédemment allouée par **malloc** ou **realloc**.