



# Libft

Ta propre bibliothèque à toi tout seul

pedago [pedago@staff.42.fr](mailto:pedago@staff.42.fr)

*Résumé: Ce projet a pour but de vous faire coder en C une bibliothèque de fonctions usuelles que vous pourrez utiliser dans tous vos projets.*

# Table des matières

<b>I</b>	<b>Préambule</b>	<b>2</b>
<b>II</b>	<b>Introduction</b>	<b>3</b>
<b>III</b>	<b>Objectifs</b>	<b>4</b>
<b>IV</b>	<b>Consignes générales</b>	<b>5</b>
<b>V</b>	<b>Partie obligatoire</b>	<b>7</b>
V.1	Considérations techniques . . . . .	7
V.2	Part 1 - Fonctions de la libc . . . . .	8
V.3	Part 2 - Fonctions supplémentaires . . . . .	9
<b>VI</b>	<b>Partie bonus</b>	<b>15</b>
<b>VII</b>	<b>Rendu et peer-évaluation</b>	<b>18</b>

# Chapitre I

## Préambule

Ce premier projet marque le début de votre formation de développeur. profitez-en pour lire [cet article](#) et apprenez aujourd'hui que le typage est ce qui différencie le développeur de la bête. Si vous ne comprenez pas tout ce n'est pas grave. Ça viendra avec le temps.

Pour vous accompagner musicalement tout au long de la réalisation de ce projet je vous propose une liste de groupes dignes d'intérêt. Si vous n'aimez pas, c'est que vous avez visiblement des goûts musicaux pauvres, mais vous avez probablement d'autres qualités comme avoir beaucoup d'amis sur Facebook ou bien pouvoir toucher votre coude avec votre langue. Bref. Les groupes sont listés sans ordre particulier et cette liste n'a pas pour but d'être exhaustive. Les liens proposés sont donnés à titre d'exemple et vous êtes encouragés à explorer vous-même leur riche discographie.

- [Between The Buried And Me](#)
- [Between The Buried And Me, c'est bon, mangez-en](#)
- [Tesseract](#)
- [Chimp Spanner](#)
- [Emancipator](#)
- [Cynic](#)
- [Kalisia](#)
- [O.S.I](#)
- [Dream Theater](#)
- [Pain Of Salvation](#)
- [Crucified Barbara](#)

# Chapitre II

## Introduction

Le projet `libft` reprend le concept du D06 de la piscine, à savoir vous faire écrire une bibliothèque de fonctions utiles que vous pourrez ensuite utiliser dans la vaste majorité de vos projets de C cette année et ainsi vous faire gagner beaucoup de temps. Ce projet vous demande d'écrire beaucoup de code que vous avez déjà réalisé pendant la piscine, ce qui en fait un excellent moment pour faire le point sur votre avancement.



FIGURE II.1 – Représentation de votre Libft (vision d'artiste)

# Chapitre III

## Objectifs

La programmation en C est une activité très laborieuse dès lors qu'on a pas accès à toutes ces petites fonctions usuelles très pratiques. C'est pourquoi nous vous proposons à travers ce projet de prendre le temps de réécrire ces fonctions, de les comprendre et de vous les approprier. Vous pourrez alors réutiliser votre bibliothèque pour travailler efficacement sur vos projets en C suivants.

Ce projet est également pour vous l'occasion d'étendre la liste des fonctions demandées avec les vôtres et ainsi de rendre votre bibliothèque encore plus utile. N'hésitez pas à compléter votre `libft` tout au long de votre scolarité une fois que ce projet ne sera plus qu'un souvenir pour vous.

# Chapitre IV

## Consignes générales

- Les fonctions sont à réaliser dans l'ordre que vous souhaitez et vous êtes très encouragés à utiliser les fonctions déjà codées pour réaliser les suivantes. La difficulté n'est pas croissante et l'ordre du sujet parfaitement arbitraire. C'est un peu comme dans un jeu vidéo où vous pouvez réaliser des quêtes dans l'ordre que vous voulez et utiliser le loot des précédentes pour vous faciliter les suivantes.
- Votre projet doit être à la Norme
- En aucun cas vos fonctions ne doivent quitter de façon inattendue (Segmentation fault, bus error, double free, etc) en dehors des comportements indéterminés. Votre projet serait alors considéré comme non fonctionnel et recevra la note de 0 en soutenance.
- Toute mémoire allouée sur le tas doit être libérée proprement quand nécessaire.
- Vous devez rendre, à la racine de votre dépôt de rendu, un fichier **auteur** contenant votre login suivi d'un `'\n'` :

```
$>cat -e auteur  
xlogin$
```

- Vous devez rendre un fichier **C** par fonction à réaliser ainsi qu'un fichier **libft.h** qui contiendra tous leurs prototypes ainsi que les **macros** et les **typedefs** dont vous pourriez avoir besoin. Tous ces fichiers devront se trouver à la racine de votre dépôt.
- Votre dossier ne devra comprendre qu'un seul **.h** qui sera **libft.h**.
- Vous devez rendre un **Makefile** qui compilera vos sources vers une bibliothèque statique nommée **libft.a**.
- Votre **Makefile** doit au moins proposer les règles **\$(NAME)**, **all**, **clean**, **fclean** et **re** dans l'ordre qui vous paraîtra le plus adapté.
- Votre **Makefile** doit compiler votre travail avec les flags de compilation **-Wall**, **-Wextra** et **-Werror**.

- Seules les fonctions suivantes de la `libc` sont autorisées : `malloc(3)`, `free(3)` et `write(2)` et leur utilisation est restreinte, voir plus bas.
- Vous devez bien entendu inclure l'`include` système nécessaire pour utiliser l'une ou l'autre des 3 fonctions autorisées dans votre fichier `.c` concerné. Le seul `include` système que vous êtes autorisés à utiliser en plus est `string.h` pour avoir accès à la constante `NULL` et au type `size_t`. Tout le reste est interdit.
- Nous vous encourageons à réaliser un ou plusieurs programmes de test pour votre bibliothèque. Bien que ce travail ne soit **pas à rendre sur votre dépôt et ne sera pas évalué**, il vous permettra de tester facilement votre travail et celui des autres.

Hormis dans le cadre de la piscine à distance qui n'en contient pas, vous trouverez une utilité particulière pour ces tests lors des soutenances. Vous êtes dans ce cadre là libres d'utiliser vos tests ou ceux du souteneur/soutenu voire même les deux si cela vous fait plaisir et la logistique sous-jacente est à votre discrétion.

# Chapitre V

## Partie obligatoire

### V.1 Considérations techniques

- Votre fichier `libft.h` peut contenir des **macros** et des **typedefs** selon vos besoins.
- Une chaîne de caractères est **TOUJOURS** terminée par un `'\0'`, même si cela a été omis dans la description d'une fonction. Dans le cas contraire, cela serait explicitement indiqué.
- Interdiction d'utiliser des variables globales.
- Si vous avez besoin de fonctions auxiliaires pour l'écriture d'une fonction complexe, vous devez définir ces fonctions auxiliaires en **static** dans le respect de la Norme.



Savoir ce qu'est une fonction statique est un bon début : <http://codingfreak.blogspot.com/2010/06/static-functions-in-c.html>

- Vous devez prêter attention à vos types et utiliser judicieusement les casts quand c'est nécessaire, en particulier lorsqu'un type `void *` est impliqué. Dans l'absolu, évitez les casts implicites, quels que soient les types concernés. Exemple :

```
char    *str;

str = malloc(42 * sizeof(*str));           /* Wrong ! Malloc retourne un void * (cast implicite) */
str = (char *) malloc(42 * sizeof(*str));   /* Right ! (cast explicite) */
```



## V.2 Part 1 - Fonctions de la libc

Dans cette première partie, vous devez recoder un ensemble de fonctions de la `libc` telles que décrites dans leur `man` respectif sur votre système. Vos fonctions devront avoir exactement le même prototype et le même comportement que les originales. Leur nom devra être préfixé par “`ft_`”. Par exemple `strlen` devient `ft_strlen`.



Certains prototypes des fonctions que vous devez recoder utilisent le qualifieur de type “`restrict`”. Ce mot clef fait parti du standard `c99`, vous devez donc ne pas le mettre dans vos prototypes et ne pas compiler avec le flag `-std=c99`.

Vous devez recoder les fonctions suivantes :

- `memset`
- `bzero`
- `memcpy`
- `memccpy`
- `memmove`
- `memchr`
- `memcmp`
- `strlen`
- `strdup`
- `strcpy`
- `strncpy`
- `strcat`
- `strncat`
- `strlcat`
- `strchr`
- `strrchr`
- `strstr`
- `strnstr`
- `strcmp`
- `strncmp`
- `atoi`
- `isalpha`
- `isdigit`
- `isalnum`
- `isascii`
- `isprint`
- `toupper`
- `tolower`

## V.3 Part 2 - Fonctions supplémentaires

Dans cette seconde partie, vous devrez coder un certain nombre de fonctions absentes de la `libc` ou présentes dans une forme différente. Certaines de ces fonctions peuvent avoir de l'intérêt pour faciliter l'écriture des fonctions de la première partie.

ft_memalloc	
<b>Prototype</b>	<code>void * ft_memalloc(size_t size);</code>
<b>Description</b>	Alloue (avec <code>malloc(3)</code> ) et retourne une zone de mémoire "fraiche". La mémoire allouée est initialisée à 0. Si l'allocation échoue, la fonction renvoie <code>NULL</code> .
<b>Param. #1</b>	La taille de la zone de mémoire à allouer.
<b>Retour</b>	La zone de mémoire allouée.
<b>Fonctions libc</b>	<code>malloc(3)</code>

ft_memdel	
<b>Prototype</b>	<code>void ft_memdel(void **ap);</code>
<b>Description</b>	Prend en paramètre l'adresse d'un pointeur dont la zone pointée doit être libérée avec <code>free(3)</code> , puis le pointeur est mis à <code>NULL</code> .
<b>Param. #1</b>	L'adresse d'un pointeur dont il faut libérer la mémoire puis le mettre à <code>NULL</code> .
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	<code>free(3)</code> .

ft_strnew	
<b>Prototype</b>	<code>char * ft_strnew(size_t size);</code>
<b>Description</b>	Alloue (avec <code>malloc(3)</code> ) et retourne une chaîne de caractère "fraiche" terminée par un <code>'\0'</code> . Chaque caractère de la chaîne est initialisé à <code>'\0'</code> . Si l'allocation échoue, la fonction renvoie <code>NULL</code> .
<b>Param. #1</b>	La taille de la chaîne de caractères à allouer.
<b>Retour</b>	La chaîne de caractères allouée et initialisée à 0.
<b>Fonctions libc</b>	<code>malloc(3)</code>

ft_strdel	
<b>Prototype</b>	<code>void ft_strdel(char **as);</code>
<b>Description</b>	Prend en paramètre l'adresse d'une chaîne de caractères qui doit être libérée avec <code>free(3)</code> et son pointeur mis à <code>NULL</code> .
<b>Param. #1</b>	L'adresse de la chaîne de caractère dont il faut libérer la mémoire et mettre le pointeur à <code>NULL</code> .
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	<code>Free(3)</code> .

ft_strclr	
<b>Prototype</b>	<code>void ft_strclr(char *s);</code>
<b>Description</b>	Assigne la valeur '\0' à tous les caractères de la chaîne passée en paramètre.
<b>Param. #1</b>	La chaîne de caractères à clearer.
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	Aucune.

ft_striter	
<b>Prototype</b>	<code>void ft_striter(char *s, void (*f)(char *));</code>
<b>Description</b>	Applique la fonction f à chaque caractère de la chaîne de caractères passée en paramètre. Chaque caractère est passé par adresse à la fonction f afin de pouvoir être modifié si nécessaire.
<b>Param. #1</b>	La chaîne de caractères sur laquelle itérer.
<b>Param. #2</b>	La fonction à appeler sur chaque caractère de s.
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	Aucune.

ft_striteri	
<b>Prototype</b>	<code>void ft_striteri(char *s, void (*f)(unsigned int, char *));</code>
<b>Description</b>	Applique la fonction f à chaque caractère de la chaîne de caractères passée en paramètre en précisant son index en premier argument. Chaque caractère est passé par adresse à la fonction f afin de pouvoir être modifié si nécessaire.
<b>Param. #1</b>	La chaîne de caractères sur laquelle itérer.
<b>Param. #2</b>	La fonction à appeler sur chaque caractère de s et son index.
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	Aucune.

ft_strmap	
<b>Prototype</b>	<code>char * ft_strmap(char const *s, char (*f)(char));</code>
<b>Description</b>	Applique la fonction f à chaque caractère de la chaîne de caractères passée en paramètre pour créer une nouvelle chaîne "fraîche" (avec malloc(3)) résultant des applications successives de f.
<b>Param. #1</b>	La chaîne de caractères sur laquelle itérer.
<b>Param. #2</b>	La fonction à appeler sur chaque caractère de s.
<b>Retour</b>	La chaîne "fraîche" résultant des applications successives de f.
<b>Fonctions libc</b>	malloc(3)

ft_strmapi	
<b>Prototype</b>	char * ft_strmapi(char const *s, char (*f)(unsigned int, char));
<b>Description</b>	Applique la fonction f à chaque caractère de la chaîne de caractères passée en paramètre en précisant son index pour créer une nouvelle chaîne "fraiche" (avec malloc(3)) résultant des applications successives de f.
<b>Param. #1</b>	La chaîne de caractères sur laquelle itérer.
<b>Param. #2</b>	La fonction à appeler sur chaque caractère de s en précisant son index.
<b>Retour</b>	La chaîne "fraiche" résultant des applications successives de f.
<b>Fonctions libc</b>	malloc(3)

ft_strequ	
<b>Prototype</b>	int ft_strequ(char const *s1, char const *s2);
<b>Description</b>	Compare lexicographiquement s1 et s2. Si les deux chaînes sont égales, la fonction retourne 1, ou 0 sinon.
<b>Param. #1</b>	La première des deux chaînes à comparer.
<b>Param. #2</b>	La seconde des deux chaînes à comparer.
<b>Retour</b>	1 ou 0 selon que les deux chaînes sont égales ou non.
<b>Fonctions libc</b>	Aucune.

ft_strnequ	
<b>Prototype</b>	int ft_strnequ(char const *s1, char const *s2, size_t n);
<b>Description</b>	Compare lexicographiquement s1 et s2 jusqu'à n caractères maximum ou bien qu'un '\0' ait été rencontré. Si les deux chaînes sont égales, la fonction retourne 1, ou 0 sinon.
<b>Param. #1</b>	La première des deux chaînes à comparer.
<b>Param. #2</b>	La seconde des deux chaînes à comparer.
<b>Param. #3</b>	Le nombre de caractères à comparer au maximum.
<b>Retour</b>	1 ou 0 selon que les deux chaînes sont égales ou non.
<b>Fonctions libc</b>	Aucune.

<b>ft_strsub</b>	
<b>Prototype</b>	<code>char * ft_strsub(char const *s, unsigned int start, size_t len);</code>
<b>Description</b>	Alloue (avec <code>malloc(3)</code> ) et retourne la copie “fraiche” d’un tronçon de la chaine de caractères passée en paramètre. Le tronçon commence à l’index <code>start</code> et à pour longueur <code>len</code> . Si <code>start</code> et <code>len</code> ne désignent pas un tronçon de chaine valide, le comportement est indéterminé. Si l’allocation échoue, la fonction renvoie <code>NULL</code> .
<b>Param. #1</b>	La chaine de caractères dans laquelle chercher le tronçon à copier.
<b>Param. #2</b>	L’index dans la chaine de caractères où débute le tronçon à copier.
<b>Param. #3</b>	La longueur du tronçon à copier.
<b>Retour</b>	Le tronçon.
<b>Fonctions libc</b>	<code>malloc(3)</code>

<b>ft_strjoin</b>	
<b>Prototype</b>	<code>char * ft_strjoin(char const *s1, char const *s2);</code>
<b>Description</b>	Alloue (avec <code>malloc(3)</code> ) et retourne une chaine de caractères “fraiche” terminée par un <code>'\0'</code> résultant de la concaténation de <code>s1</code> et <code>s2</code> . Si l’allocation échoue, la fonction renvoie <code>NULL</code> .
<b>Param. #1</b>	La chaine de caractères préfixe.
<b>Param. #2</b>	La chaine de caractères suffixe.
<b>Retour</b>	La chaine de caractère “fraiche” résultant de la concaténation des deux chaines.
<b>Fonctions libc</b>	<code>malloc(3)</code>

<b>ft_strtrim</b>	
<b>Prototype</b>	<code>char * ft_strtrim(char const *s);</code>
<b>Description</b>	Alloue (avec <code>malloc(3)</code> ) et retourne une copie de la chaine passée en paramètre sans les espaces blancs au debut et à la fin de cette chaine. On considère comme espaces blancs les caractères <code>' '</code> , <code>'\n'</code> et <code>'\t'</code> . Si <code>s</code> ne contient pas d’espaces blancs au début ou à la fin, la fonction renvoie une copie de <code>s</code> . Si l’allocation échoue, la fonction renvoie <code>NULL</code> .
<b>Param. #1</b>	La chaine de caractères à trimmer.
<b>Retour</b>	La chaine de caractère “fraiche” trimmée ou bien une copie de <code>s</code> sinon.
<b>Fonctions libc</b>	<code>malloc(3)</code>

ft_strsplit	
<b>Prototype</b>	<code>char ** ft_strsplit(char const *s, char c);</code>
<b>Description</b>	Alloue (avec <code>malloc(3)</code> ) et retourne un tableau de chaînes de caractères “fraîches” (toutes terminées par un <code>'\0'</code> , le tableau également donc) résultant de la découpe de <code>s</code> selon le caractère <code>c</code> . Si l'allocation échoue, la fonction retourne <code>NULL</code> . Exemple : <code>ft_strsplit("salut les étudiants", '*')</code> renvoie le tableau <code>["salut", "les", "etudiants"]</code> .
<b>Param. #1</b>	La chaîne de caractères à découper.
<b>Param. #2</b>	Le caractère selon lequel découper la chaîne.
<b>Retour</b>	Le tableau de chaînes de caractères “fraîches” résultant de la découpe.
<b>Fonctions libc</b>	<code>malloc(3)</code> , <code>free(3)</code>

ft_itoa	
<b>Prototype</b>	<code>char * ft_itoa(int n);</code>
<b>Description</b>	Alloue (avec <code>malloc(3)</code> ) et retourne une chaîne de caractères “fraîche” terminée par un <code>'\0'</code> représentant l'entier <code>n</code> passé en paramètre. Les nombres négatifs doivent être gérés. Si l'allocation échoue, la fonction renvoie <code>NULL</code> .
<b>Param. #1</b>	L'entier à convertir en une chaîne de caractères.
<b>Retour</b>	La chaîne de caractères représentant l'entier passé en paramètre.
<b>Fonctions libc</b>	<code>malloc(3)</code>

ft_putchar	
<b>Prototype</b>	<code>void ft_putchar(char c);</code>
<b>Description</b>	Affiche le caractère <code>c</code> sur la sortie standard.
<b>Param. #1</b>	Le caractère à afficher.
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	<code>write(2)</code> .

ft_putstr	
<b>Prototype</b>	<code>void ft_putstr(char const *s);</code>
<b>Description</b>	Affiche la chaîne <code>s</code> sur la sortie standard.
<b>Param. #1</b>	La chaîne de caractères à afficher.
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	<code>write(2)</code> .

<b>ft_putendl</b>	
<b>Prototype</b>	<code>void ft_putendl(char const *s);</code>
<b>Description</b>	Affiche la chaîne <code>s</code> sur la sortie standard suivi d'un <code>'\n'</code> .
<b>Param. #1</b>	La chaîne de caractères à afficher.
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	<code>write(2)</code> .

<b>ft_putnbr</b>	
<b>Prototype</b>	<code>void ft_putnbr(int n);</code>
<b>Description</b>	Affiche l'entier <code>n</code> sur la sortie standard.
<b>Param. #1</b>	L'entier à afficher.
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	<code>write(2)</code> .

<b>ft_putchar_fd</b>	
<b>Prototype</b>	<code>void ft_putchar_fd(char c, int fd);</code>
<b>Description</b>	Ecrit le caractère <code>c</code> sur le descripteur de fichier <code>fd</code> .
<b>Param. #1</b>	Le caractères à écrire.
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	<code>write(2)</code> .

<b>ft_putstr_fd</b>	
<b>Prototype</b>	<code>void ft_putstr_fd(char const *s, int fd);</code>
<b>Description</b>	Ecrit la chaîne <code>s</code> sur le descripteur de fichier <code>fd</code> .
<b>Param. #1</b>	La chaîne de caractères à écrire.
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	<code>write(2)</code> .

<b>ft_putendl_fd</b>	
<b>Prototype</b>	<code>void ft_putendl_fd(char const *s, int fd);</code>
<b>Description</b>	Ecrit la chaîne <code>s</code> sur le descripteur de fichier <code>fd</code> suivi d'un <code>'\n'</code> .
<b>Param. #1</b>	La chaîne de caractères à écrire.
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	<code>write(2)</code> .

<b>ft_putnbr_fd</b>	
<b>Prototype</b>	<code>void ft_putnbr_fd(int n, int fd);</code>
<b>Description</b>	Ecrit l'entier <code>n</code> sur le descripteur de fichier <code>fd</code> .
<b>Param. #1</b>	L'entier à écrire.
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	<code>write(2)</code> .

# Chapitre VI

## Partie bonus

Si vous avez réussi parfaitement la partie obligatoire, cette section propose quelques pistes pour aller plus loin. Un peu comme quand vous achetez un DLC pour un jeu vidéo.

Avoir des fonctions de manipulation de mémoire brute et de chaînes de caractères est très pratique, mais vous vous rendrez vite compte qu'avoir des fonctions de manipulation de liste est encore plus pratique.

Vous utiliserez la structure suivante pour représenter les maillons de votre liste. Cette structure est à ajouter à votre fichier `libft.h`.

```
typedef struct    s_list
{
    void          *content;
    size_t        content_size;
    struct s_list *next;
} t_list;
```

La description des champs de la structure `t_list` est la suivante :

- **content** : La donnée contenue dans le maillon. Le `void *` permet de stocker une donnée de n'importe quel type.
- **content\_size** : La taille de la donnée stockée. Le type `void *` ne permettant pas de connaître la taille de la donnée pointée, il est nécessaire d'en sauvegarder la taille. Par exemple la chaîne de caractères "42" a une taille de 3 octets et l'entier 32bits 42 a une taille de 4 octets.
- **next** : L'adresse du maillon suivant de la liste ou `NULL` si le maillon est le dernier.



Les fonctions suivantes vous permettront de manipuler vos listes aisément.

ft_lstnew	
<b>Prototype</b>	<code>t_list * ft_lstnew(void const *content, size_t content_size);</code>
<b>Description</b>	Alloue (avec <code>malloc(3)</code> ) et retourne un maillon “frais”. Les champs <code>content</code> et <code>content_size</code> du nouveau maillon sont initialisés par <b>copie</b> des paramètres de la fonction. Si le paramètre <code>content</code> est nul, le champs <code>content</code> est initialisé à <code>NULL</code> et le champs <code>content_size</code> est initialisé à 0 quelque soit la valeur du paramètre <code>content_size</code> . Le champ <code>next</code> est initialisé à <code>NULL</code> . Si l’allocation échoue, la fonction renvoie <code>NULL</code> .
<b>Param. #1</b>	Le contenu à ajouter au nouveau maillon.
<b>Param. #2</b>	La taille du contenu à ajouter au nouveau maillon.
<b>Retour</b>	Le nouveau maillon.
<b>Fonctions libc</b>	<code>malloc(3)</code> , <code>free(3)</code>

ft_lstdelone	
<b>Prototype</b>	<code>void ft_lstdelone(t_list **alst, void (*del)(void *, size_t));</code>
<b>Description</b>	Prend en paramètre l’adresse d’un pointeur sur un maillon et libère la mémoire du contenu de ce maillon avec la fonction <code>del</code> passée en paramètre puis libère la mémoire du maillon en lui même avec <code>free(3)</code> . La mémoire du champ <code>next</code> ne doit en aucun cas être libérée. Pour terminer, le pointeur sur le maillon maintenant libéré doit être mis à <code>NULL</code> (de manière similaire à la fonction <code>ft_memdel</code> de la partie obligatoire).
<b>Param. #1</b>	L’adresse d’un pointeur sur le maillon à libérer.
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	<code>free(3)</code>

ft_lstdel	
<b>Prototype</b>	<code>void ft_lstdel(t_list **alst, void (*del)(void *, size_t));</code>
<b>Description</b>	Prend en paramètre l’adresse d’un pointeur sur un maillon et libère la mémoire de ce maillon et celle de tous ses successeurs l’un après l’autre avec <code>del</code> et <code>free(3)</code> . Pour terminer, le pointeur sur le premier maillon maintenant libéré doit être mis à <code>NULL</code> (de manière similaire à la fonction <code>ft_memdel</code> de la partie obligatoire).
<b>Param. #1</b>	L’adresse d’un pointeur sur le premier maillon d’une liste à libérer.
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	<code>free(3)</code>

ft_lstadd	
<b>Prototype</b>	<code>void ft_lstadd(t_list **alst, t_list *new);</code>
<b>Description</b>	Ajoute l'élément <code>new</code> en tête de la liste.
• <b>Param. #1</b>	L'adresse d'un pointeur sur le premier maillon d'une liste.
<b>Param. #2</b>	Le maillon à ajouter en tête de cette liste.
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	Aucune.

ft_lstiter	
<b>Prototype</b>	<code>void ft_lstiter(t_list *lst, void (*f)(t_list *elem));</code>
<b>Description</b>	Parcourt la liste <code>lst</code> en appliquant à chaque maillon la fonction <code>f</code> .
• <b>Param. #1</b>	Pointeur sur le premier maillon d'une liste.
<b>Param. #2</b>	L'adresse d'une fonction à laquelle appliquer chaque maillon de la liste.
<b>Retour</b>	Rien.
<b>Fonctions libc</b>	Aucune.

ft_lstmap	
<b>Prototype</b>	<code>t_list * ft_lstmap(t_list *lst, t_list * (*f)(t_list *elem));</code>
<b>Description</b>	Parcourt la liste <code>lst</code> en appliquant à chaque maillon la fonction <code>f</code> et crée une nouvelle liste "fraîche" avec <code>malloc(3)</code> résultant des applications successives. Si une allocation échoue, la fonction renvoie <code>NULL</code> .
• <b>Param. #1</b>	Pointeur sur le premier maillon d'une liste.
<b>Param. #2</b>	L'adresse d'une fonction à appliquer à chaque maillon de la liste pour créer une nouvelle liste.
<b>Retour</b>	La nouvelle liste.
<b>Fonctions libc</b>	<code>malloc(3)</code> , <code>free(3)</code> .

Si vous réussissez parfaitement la partie obligatoire et la partie bonus, vous êtes encouragés à ajouter d'autres fonctions qui vous paraissent utiles pour agrandir votre bibliothèque. Exemples : une version de `ft_strsplit` qui renvoie une liste de chaînes au lieu d'un tableau de chaînes, la fonction `ft_lstfold` similaire à la fonction `reduce` de `Python` et à la fonction `List.fold_left` d'`OCaml` (attention aux fuites mémoires!), des fonctions de manipulation de tableaux, de piles, de files, de maps, de tables de hash, etc. La limite est votre imagination.

# Chapitre VII

## Rendu et peer-évaluation

Rendez-votre travail sur votre dépôt GiT comme d'habitude. Seul le travail présent sur votre dépôt sera évalué.

**Si et seulement si vous réalisez ce projet dans le cadre de la piscine à distance :** Votre travail sera évalué par une moulinette uniquement.

**Sinon :** Une fois vos soutenances terminées une moulinette passera sur votre rendu. Votre note finale sera calculée en prenant en compte les notes que vous avez reçues en peer-évaluation et la note de la moulinette.

La moulinette corrigera tous vos exercices en suivant le schéma donné dans ce sujet (part1, part2, bonus). Une erreur dans l'une des parties entraînera systématiquement l'arrêt de la comptabilisation des points des parties suivantes.

Bon courage à tous et n'oubliez pas votre fichier auteur !