Chapter V

Mandatory part

V.1 Technical considerations

- Your `libft.h` file can contain `macros` and `typedefs` if needed.
- A string must **ALWAYS** end with a `'\0'`, even if it is not included in the function's description, unless explicitly stated otherwise.
- It is forbidden to use global variables.
- If you need sub-functions to write a complex function, you must define these sub-functions as `static` as stipulated in the Norm.

`Check out this link to find out more about static functions:`
`http://codingfreak.blogspot.com/2010/06/static-functions-in-c.html`

- You must pay attention to your types and wisely use the casts when needed, especially when a `void*` type is involved. Generally speaking, avoid implicit casts. Example:

```c
char    *str;

str = malloc(42 * sizeof(*str));            /* Wrong ! Malloc returns a void * (implicit cast) */
str = (char *) malloc(42 * sizeof(*str));   /* Right ! (explicit cast) */
```

V.2 Part 1 - Libc functions

In this first part, you must re-code a set of the `libc` functions, as defined in their `man`. Your functions will need to present the same prototype and behaviors as the originals. Your functions' names must be prefixed by "`ft_`". For instance `strlen` becomes `ft_strlen`.

```
Some of the functions' prototypes you have to re-code use the
"restrict" qualifier.  This keyword is part of the c99 standard.
It is therefore forbidden to include it in your prototypes and to
compile it with the flag -std=c99.
```

You must re-code the following functions:

- `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 - Additional functions

In this second part, you must code a set of functions that are either not included in the `libc`, or included in a different form. Some of these functions can be useful to write Part 1's functions.

-

ft_memalloc	
Prototype	`void * ft_memalloc(size_t size);`
Description	Allocates (with `malloc(3)`) and returns a "fresh" memory area. The memory allocated is initialized to `0`. If the allocation fails, the function returns `NULL`.
Param. #1	The size of the memory that needs to be allocated.
Return value	The allocated memory area.
Libc functions	`malloc(3)`

-

ft_memdel	
Prototype	`void ft_memdel(void **ap);`
Description	Takes as a parameter the address of a memory area that needs to be freed with `free(3)`, then puts the pointer to `NULL`.
Param. #1	A pointer's address that needs its memory freed and set to `NULL`.
Return value	None.
Libc functions	`free(3)`.

-

ft_strnew	
Prototype	`char * ft_strnew(size_t size);`
Description	Allocates (with `malloc(3)`) and returns a "fresh" string ending with `'\0'`. Each character of the string is initialized at `'\0'`. If the allocation fails the function returns `NULL`.
Param. #1	The size of the string to be allocated.
Return value	The string allocated and initialized to `0`.
Libc functions	`malloc(3)`

-

ft_strdel	
Prototype	`void ft_strdel(char **as);`
Description	Takes as a parameter the address of a string that need to be freed with `free(3)`, then sets its pointer to `NULL`.
Param. #1	The string's address that needs to be freed and its pointer set to `NULL`.
Return value	None.
Libc functions	`Free(3)`.

-

ft_strclr	
Prototype	`void ft_strclr(char *s);`
Description	Sets every character of the string to the value `'\0'`.
Param. #1	The string that needs to be cleared.
Return value	None.
Libc functions	None.

-

ft_striter	
Prototype	`void  ft_striter(char *s, void (*f)(char *));`
Description	Applies the function `f` to each character of the string passed as argument. Each character is passed by address to `f` to be modified if necessary.
Param. #1	The string to iterate.
Param. #2	The function to apply to each character of `s`.
Return value	None.
Libc functions	None.

-

ft_striteri	
Prototype	`void  ft_striteri(char *s, void (*f)(unsigned int, char *));`
Description	Applies the function `f` to each character of the string passed as argument, and passing its index as first argument. Each character is passed by address to `f` to be modified if necessary.
Param. #1	The string to iterate.
Param. #2	The function to apply to each character of `s` and its index.
Return value	None.
Libc functions	None.

-

ft_strmap	
Prototype	`char *  ft_strmap(char const *s, char (*f)(char));`
Description	Applies the function `f` to each character of the string given as argument to create a “fresh” new string (with `malloc(3)`) resulting from the successive applications of `f`.
Param. #1	The string to map.
Param. #2	The function to apply to each character of `s`.
Return value	The “fresh” string created from the successive applications of `f`.
Libc functions	`malloc(3)`

-

ft_strmapi	
Prototype	`char *  ft_strmapi(char const *s, char (*f)(unsigned int, char));`
Description	Applies the function `f` to each character of the string passed as argument by giving its index as first argument to create a “fresh” new string (with `malloc(3)`) resulting from the successive applications of `f`.
Param. #1	The string to map.
Param. #2	The function to apply to each character of `s` and its index.
Return value	The “fresh” string created from the successive applications of `f`.
Libc functions	`malloc(3)`

•

ft_strequ	
Prototype	`int  ft_strequ(char const *s1, char const *s2);`
Description	Lexicographical comparison between `s1` and `s2`. If the 2 strings are identical the function returns `1`, or `0` otherwise.
Param. #1	The first string to be compared.
Param. #2	The second string to be compared.
Return value	`1` or `0` according to if the 2 strings are identical or not.
Libc functions	None.

•

ft_strnequ	
Prototype	`int  ft_strnequ(char const *s1, char const *s2, size_t n);`
Description	Lexicographical comparison between `s1` and `s2` up to `n` characters or until a `'\0'` is reached. If the 2 strings are identical, the function returns `1`, or `0` otherwise.
Param. #1	The first string to be compared.
Param. #2	The second string to be compared.
Param. #3	The maximum number of characters to be compared.
Return value	`1` or `0` according to if the 2 strings are identical or not.
Libc functions	None.

•

ft_strsub	
Prototype	`char *  ft_strsub(char const *s, unsigned int start, size_t len);`
Description	Allocates (with `malloc(3)`) and returns a "fresh" substring from the string given as argument. The substring begins at index`start` and is of size `len`. If `start` and `len` aren't refering to a valid substring, the behavior is undefined. If the allocation fails, the function returns `NULL`.
Param. #1	The string from which create the substring.
Param. #2	The start index of the substring.
Param. #3	The size of the substring.
Return value	The substring.
Libc functions	`malloc(3)`

•

ft_strjoin	
Prototype	`char *  ft_strjoin(char const *s1, char const *s2);`
Description	Allocates (with `malloc(3)`) and returns a "fresh" string ending with `'\0'`, result of the concatenation of `s1` and `s2`. If the allocation fails the function returns `NULL`.
Param. #1	The prefix string.
Param. #2	The suffix string.
Return value	The "fresh" string result of the concatenation of the 2 strings.
Libc functions	`malloc(3)`

ft_strtrim	
Prototype	`char * ft_strtrim(char const *s);`
Description	Allocates (with `malloc(3)`) and returns a copy of the string given as argument without whitespaces at the beginning or at the end of the string. Will be considered as whitespaces the following characters `' '`, `'\n'` and `'\t'`. If `s` has no whitespaces at the beginning or at the end, the function returns a copy of `s`. If the allocation fails the function returns `NULL`.
Param. #1	The string to be trimed.
Return value	The “fresh” trimmed string or a copy of `s`.
Libc functions	`malloc(3)`

ft_strsplit	
Prototype	`char ** ft_strsplit(char const *s, char c);`
Description	Allocates (with `malloc(3)`) and returns an array of “fresh” strings (all ending with `'\0'`, including the array itself) obtained by spliting `s` using the character `c` as a delimiter. If the allocation fails the function returns `NULL`. Example : `ft_strsplit("*hello*fellow***students*", '*')` returns the array `["hello", "fellow", "students"]`.
Param. #1	The string to split.
Param. #2	The delimiter character.
Return value	The array of “fresh” strings result of the split.
Libc functions	`malloc(3)`, `free(3)`

ft_itoa	
Prototype	`char * ft_itoa(int n);`
Description	Allocate (with `malloc(3)`) and returns a “fresh” string ending with `'\0'` representing the integer `n` given as argument. Negative numbers must be supported. If the allocation fails, the function returns `NULL`.
Param. #1	The integer to be transformed into a string.
Return value	The string representing the integer passed as argument.
Libc functions	`malloc(3)`

ft_putchar	
Prototype	`void ft_putchar(char c);`
Description	Outputs the character `c` to the standard output.
Param. #1	The character to output.
Return value	None.
Libc functions	`write(2)`.

ft_putstr	
Prototype	`void ft_putstr(char const *s);`
Description	Outputs the string `s` to the standard output.
Param. #1	The string to output.
Return value	None.
Libc functions	`write(2)`.

ft_putendl	
Prototype	`void  ft_putendl(char const *s);`
Description	Outputs the string `s` to the standard output followed by a `'\n'`.
Param. #1	The string to output.
Return value	None.
Libc functions	`write(2)`.

ft_putnbr	
Prototype	`void  ft_putnbr(int n);`
Description	Outputs the integer `n` to the standard output.
Param. #1	The integer to output.
Return value	None.
Libc functions	`write(2)`.

ft_putchar_fd	
Prototype	`void  ft_putchar_fd(char c, int fd);`
Description	Outputs the char `c` to the file descriptor `fd`.
Param. #1	The character to output.
Param. #2	The file descriptor.
Return value	None.
Libc functions	`write(2)`.

ft_putstr_fd	
Prototype	`void  ft_putstr_fd(char const *s, int fd);`
Description	Outputs the string `s` to the file descriptor `fd`.
Param. #1	The string to output.
Param. #2	The file descriptor.
Return value	None.
Libc functions	`write(2)`.

ft_putendl_fd	
Prototype	`void  ft_putendl_fd(char const *s, int fd);`
Description	Outputs the string `s` to the file descriptor `fd` followed by a `'\n'`.
Param. #1	The string to output.
Param. #2	The file descriptor.
Return value	None.
Libc functions	`write(2)`.

ft_putnbr_fd	
Prototype	`void  ft_putnbr_fd(int n, int fd);`
Description	Outputs the integer `n` to the file descriptor `fd`.
Param. #1	The integer to print.
Param. #2	The file descriptor.
Return value	None.
Libc functions	`write(2)`.

Chapter VI

Bonus part

If you successfully completed the mandatory part, you'll enjoy taking it further. You can see this last section as Bonus Points.

Having functions to manipulate memory and strings is very useful, but you'll soon discover that having functions to manipulate lists is even more useful.

You'll use the following structure to represent the links of your list. This structure must be added to your `libft.h` file.

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

Here is a description of the fields of the `t_list` struct:

- `content` : The data contained in the link. The `void *` allows to store any kind of data.
- `content_size` : The size of the data stored. The `void *` type doesn't allow you to know the size of the pointed data, as a consequence, it is necessary to save its size. For instance, the size of the string `"42"` is `3 bytes` and the 32bits integer `42` has a size of `4 bytes`.
- `next` : The next link's address or `NULL` if it's the last link.

The following functions will allow you to manipulate your lists more easilly.

ft_lstnew	
Prototype	`t_list * ft_lstnew(void const *content, size_t content_size);`
Description	Allocates (with `malloc(3)`) and returns a "fresh" link. The variables `content` and `content_size` of the new link are initialized by **copy** of the parameters of the function. If the parameter `content` is nul, the variable `content` is initialized to `NULL` and the variable `content_size` is initialized to `0` even if the parameter `content_size` isn't. The variable `next` is initialized to `NULL`. If the allocation fails, the function returns `NULL`.
Param. #1	The content to put in the new link.
Param. #2	The size of the content of the new link.
Return value	The new link.
Libc functions	`malloc(3)`, `free(3)`

ft_lstdelone	
Prototype	`void ft_lstdelone(t_list **alst, void (*del)(void *, size_t));`
Description	Takes as a parameter a link's pointer address and frees the memory of the link's content using the function `del` given as a parameter, then frees the link's memory using `free(3)`. The memory of `next` musnt not be freed under any circumstance. Finally, the pointer to the link that was just freed must be set to `NULL` (quite similar to the function `ft_memdel` in the mandatory part).
Param. #1	The adress of a pointer to a link that needs to be freed.
Return value	None.
Libc functions	`free(3)`

ft_lstdel	
Prototype	`void ft_lstdel(t_list **alst, void (*del)(void *, size_t));`
Description	Takes as a parameter the adress of a pointer to a link and frees the memory of this link and every successors of that link using the functions `del` and `free(3)`. Finally the pointer to the link that was just freed must be set to `NULL` (quite similar to the function `ft_memdel` from the mandatory part).
Param. #1	The address of a pointer to the first link of a list that needs to be freed.
Return value	None.
Libc functions	`free(3)`

ft_lstadd	
Prototype	`void ft_lstadd(t_list **alst, t_list *new);`
Description	Adds the element `new` at the beginnning of the list.
Param. #1	The address of a pointer to the first link of a list.
Param. #2	The link to add at the beginning of the list.
Return value	None.
Libc functions	None.

ft_lstiter	
Prototype	`void ft_lstiter(t_list *lst, void (*f)(t_list *elem));`
Description	Iterates the list `lst` and applies the function `f` to each link.
Param. #1	A pointer to the first link of a list.
Param. #2	The address of a function to apply to each link of a list.
Return value	None.
Libc functions	None.

ft_lstmap	
Prototype	`t_list * ft_lstmap(t_list *lst, t_list * (*f)(t_list *elem));`
Description	Iterates a list `lst` and applies the function `f` to each link to create a “fresh” list (using `malloc(3)`) resulting from the successive applications of `f`. If the allocation fails, the function returns `NULL`.
Param. #1	A pointer’s to the first link of a list.
Param. #2	The address of a function to apply to each link of a list.
Return value	The new list.
Libc functions	`malloc(3)`, `free(3)`.

If you successfully completed both the mandatory and bonus sections of this project, we encourage you to add other functions that you believe could be useful to expand your library. For instance, a version of `ft_strsplit` that returns a list instead of an array, the function `ft_lstfold` similar to the function `reduce` in `Python` and the function `List.fold_left` in `OCaml` (beware of the memory leak !). You can add functions to manipulate arrays, stacks, files, maps, hashtables, etc. The limit is your imagination.