Libft

Your very first own library

Summary:
This project involves coding a `C` *library that will include numerous general purpose functions for your programs.*

Version: 16.7

Chapter I

Introduction

C programming can be quite tedious without access to the highly useful standard functions. This project aims to help you understand how these functions work by implementing them yourself and learning to use them effectively. You will create your own library, which will be valuable for your future C school assignments.

Take the time to expand your `libft` throughout the year. However, when working on a new project, always check that the functions used in your library comply with the project guidelines.

Chapter II

Common Instructions

- Your project must be written in C.
- Your project must be written in accordance with the Norm. If you have bonus files/functions, they are included in the norm check, and you will receive a `0` if there is a norm error.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc.) except for undefined behavior. If this occurs, your project will be considered non-functional and will receive a `0` during the evaluation.
- All heap-allocated memory must be properly freed when necessary. Memory leaks will not be tolerated.
- If the subject requires it, you must submit a `Makefile` that compiles your source files to the required output with the flags `-Wall`, `-Wextra`, and `-Werror`, using `cc`. Additionally, your `Makefile` must not perform unnecessary relinking.
- Your `Makefile` must at contain at least the rules `$(NAME)`, `all`, `clean`, `fclean` and `re`.
- To submit bonuses for your project, you must include a `bonus` rule in your `Makefile`, which will add all the various headers, libraries, or functions that are not allowed in the main part of the project. Bonuses must be placed in `_bonus.{c/h}` files, unless the subject specifies otherwise. The evaluation of mandatory and bonus parts is conducted separately.
- If your project allows you to use your `libft`, you must copy its sources and its associated `Makefile` into a `libft` folder. Your project's `Makefile` must compile the library by using its `Makefile`, then compile the project.
- We encourage you to create test programs for your project, even though this work **does not need to be submitted and will not be graded**. It will give you an opportunity to easily test your work and your peers' work. You will find these tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to the assigned Git repository. Only the work in the Git repository will be graded. If Deepthought is assigned to grade your work, it will occur

after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

Chapter III

Mandatory part

Program name	`libft.a`
Turn in files	`Makefile, libft.h, ft_*.c`
Makefile	`NAME, all, clean, fclean, re`
External functs.	`Detailed below`
Libft authorized	`n/a`
Description	`Create your own library: a collection of functions that will serve as a useful tool throughout your cursus.`

III.1 Technical considerations

- Declaring global variables is strictly forbidden.
- If you need helper functions to break down a more complex function, define them as `static` functions to restrict their scope to the appropriate file.
- All files must be placed at the root of your repository.
- Submitting unused files is not allowed.
- Every `.c` file must compile with the following flags: `-Wall -Wextra -Werror`.
- You must use the `ar` command to create your library. The use of `libtool` is strictly forbidden.
- Your `libft.a` must be created at the root of your repository.

III.2 Part 1 - Libc functions

To begin, you must reimplement a set of functions from the `libc`. Your version will have the same prototypes and behaviors as the originals, adhering strictly to their definitions in the `man` page. The only difference will be their names, as they must start with the '`ft_`' prefix. For example, `strlen` becomes `ft_strlen`.

> Some of the function prototypes you need to reimplement use the 'restrict' qualifier. This keyword is part of the C99 standard. Therefore, it is forbidden to include it in your own prototypes or to compile your code with the -std=c99 flag.

The following functions must be rewritten without relying on external functions:

- `isalpha`
- `isdigit`
- `isalnum`
- `isascii`
- `isprint`
- `strlen`
- `memset`
- `bzero`
- `memcpy`
- `memmove`
- `strlcpy`
- `strlcat`
- `toupper`
- `tolower`
- `strchr`
- `strrchr`
- `strncmp`
- `memchr`
- `memcmp`
- `strnstr`
- `atoi`

To implement the two following functions, you will use `malloc()`:

- `calloc`
- `strdup`

> Depending on your current operating system, the 'calloc' function's behavior may differ from its man page description. Follow this rule instead: If nmemb or size is 0, then calloc() returns a unique pointer value that can be successfully passed to free().

III.3 Part 2 - Additional functions

In this second part, you must develop a set of functions that are either not included in the `libc`, or exist in a different form.

> Some of the functions from Part 1 may be useful for implementing the functions below.

Function name	ft_substr
Prototype	`char *ft_substr(char const *s, unsigned int start, size_t len);`
Turn in files	-
Parameters	s: The original string from which to create the substring. start: The starting index of the substring within 's'. len: The maximum length of the substring.
Return value	The substring. NULL if the allocation fails.
External functs.	malloc
Description	Allocates memory (using malloc(3)) and returns a substring from the string 's'. The substring starts at index 'start' and has a maximum length of 'len'.

Function name	ft_strjoin
Prototype	`char *ft_strjoin(char const *s1, char const *s2);`
Turn in files	-
Parameters	s1: The prefix string. s2: The suffix string.
Return value	The new string. NULL if the allocation fails.
External functs.	malloc
Description	Allocates memory (using malloc(3)) and returns a new string, which is the result of concatening 's1' and 's2'.

Function name	`ft_strtrim`
Prototype	`char *ft_strtrim(char const *s1, char const *set);`
Turn in files	-
Parameters	s1: The string to be trimmed. set: The string containing the set of characters to be removed.
Return value	The trimmed string. NULL if the allocation fails.
External functs.	malloc
Description	Allocates memory (using malloc(3)) and returns a copy of 's1' with characters from 'set' removed from the beginning and the end.

Function name	`ft_split`
Prototype	`char **ft_split(char const *s, char c);`
Turn in files	-
Parameters	s: The string to be split. c: The delimiter character.
Return value	The array of new strings resulting from the split. NULL if the allocation fails.
External functs.	malloc, free
Description	Allocates memory (using malloc(3)) and returns an array of strings obtained by splitting 's' using the character 'c' as a delimiter. The array must end with a NULL pointer.

Function name	`ft_itoa`
Prototype	`char *ft_itoa(int n);`
Turn in files	-
Parameters	n: The integer to convert.
Return value	The string representing the integer. NULL if the allocation fails.
External functs.	malloc
Description	Allocates memory (using malloc(3)) and returns a string representing the integer received as an argument. Negative numbers must be handled.

Function name	ft_strmapi
Prototype	char *ft_strmapi(char const *s, char (*f)(unsigned int, char));
Turn in files	-
Parameters	s: The string to iterate over. f: The function to apply to each character.
Return value	The string created from the successive applications of ’f’. Returns NULL if the allocation fails.
External functs.	malloc
Description	Applies the function f to each character of the string s, passing its index as the first argument and the character itself as the second. A new string is created (using malloc(3)) to store the results from the successive applications of f.

Function name	ft_striteri
Prototype	void ft_striteri(char *s, void (*f)(unsigned int, char*));
Turn in files	-
Parameters	s: The string to iterate over. f: The function to apply to each character.
Return value	None
External functs.	None
Description	Applies the function ’f’ to each character of the string passed as argument, passing its index as the first argument. Each character is passed by address to ’f’ so it can be modified if necessary.

Function name	ft_putchar_fd
Prototype	void ft_putchar_fd(char c, int fd);
Turn in files	-
Parameters	c: The character to output. fd: The file descriptor on which to write.
Return value	None
External functs.	write
Description	Outputs the character ’c’ to the specified file descriptor.

Function name	`ft_putstr_fd`
Prototype	`void ft_putstr_fd(char *s, int fd);`
Turn in files	-
Parameters	s: The string to output. fd: The file descriptor on which to write.
Return value	None
External functs.	write
Description	Outputs the string ’s’ to the specified file descriptor.

Function name	`ft_putendl_fd`
Prototype	`void ft_putendl_fd(char *s, int fd);`
Turn in files	-
Parameters	s: The string to output. fd: The file descriptor on which to write.
Return value	None
External functs.	write
Description	Outputs the string ’s’ to the specified file descriptor followed by a newline.

Function name	`ft_putnbr_fd`
Prototype	`void ft_putnbr_fd(int n, int fd);`
Turn in files	-
Parameters	n: The integer to output. fd: The file descriptor on which to write.
Return value	None
External functs.	write
Description	Outputs the integer ’n’ to the specified file descriptor.