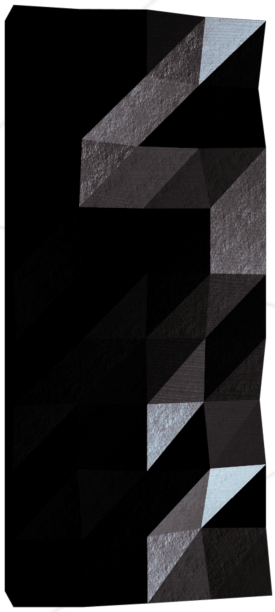

Libft

Your very first own library

Summary:
This project is about coding a `C` library.
It will contain a lot of general purpose functions your programs will rely upon.

Version: 15

Contents

Chapter I

Introduction

`C` programming can be very tedious when one doesn't have access to the highly useful standard functions. This project is about understanding the way these functions work, implementing and learning to use them. Your will create your own library. It will be helpful since you will use it in your next `C` school assignments.

Take the time to expand your `libft` throughout the year. However, when working on a new project, don't forget to ensure the functions used in your library are allowed in 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 inside.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc) apart from undefined behaviors. If this happens, your project will be considered non functional and will receive a `0` during the evaluation.
- All heap allocated memory space must be properly freed when necessary. No leaks will be tolerated.
- If the subject requires it, you must submit a `Makefile` which will compile your source files to the required output with the flags `-Wall`, `-Wextra` and `-Werror`, use cc, and your Makefile must not relink.
- Your `Makefile` must at least contain the rules `$(NAME)`, `all`, `clean`, `fclean` and `re`.
- To turn in bonuses to your project, you must include a rule `bonus` to your Makefile, which will add all the various headers, librairies or functions that are forbidden on the main part of the project. Bonuses must be in a different file `_bonus.{c/h}` if the subject does not specify anything else. Mandatory and bonus part evaluation is done separately.
- If your project allows you to use your `libft`, you must copy its sources and its associated `Makefile` in a `libft` folder with its associated Makefile. 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 **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those 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 your assigned git repository. Only the work in the git repository will be graded. If Deepthought is assigned to grade your work, it will be done

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	`Write your own library: a collection of functions that will be a useful tool for your cursus.`

III.1 Technical considerations

- Declaring global variables is forbidden.
- If you need helper functions to split a more complex function, define them as `static` functions. This way, their scope will be limited to the appropriate file.
- Place all your files at the root of your repository.
- Turning in unused files is forbidden.
- Every `.c` files must compile with the flags `-Wall -Wextra -Werror`.
- You must use the command `ar` to create your library. Using the `libtool` command is forbidden.
- Your `libft.a` has to be created at the root of your repository.

III.2 Part 1 - Libc functions

To begin, you must redo a set of functions from the `libc`. Your functions will have the same prototypes and implement the same behaviors as the originals. They must comply with the way they are defined in their `man`. The only difference will be their names. They will begin with the 'ft_' prefix. For instance, `strlen` becomes `ft_strlen`.

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

You must write your own function implementing the following original ones. They do not require any external functions:

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

In order to implement the two following functions, you will use `malloc()`:

- calloc
- strdup

III.3 Part 2 - Additional functions

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

Some of the following functions can be useful for writing the functions of Part 1.

Function name	ft_substr
Prototype	char *ft_substr(char const *s, unsigned int start, size_t len);
Turn in files	-
Parameters	s: The string from which to create the substring. start: The start index of the substring in the string 's'. len: The maximum length of the substring.
Return value	The substring. NULL if the allocation fails.
External functs.	malloc
Description	Allocates (with malloc(3)) and returns a substring from the string 's'. The substring begins at index 'start' and is of maximum size '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 (with malloc(3)) and returns a new string, which is the result of the concatenation of '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 reference set of characters to trim.
Return value	The trimmed string. NULL if the allocation fails.
External functs.	malloc
Description	Allocates (with malloc(3)) and returns a copy of 's1' with the characters specified in 'set' removed from the beginning and the end of the string.

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 (with 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 (with 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 on which to iterate. 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', and passing its index as first argument to create a new string (with malloc(3)) resulting from 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 on which to iterate. f: The function to apply to each character.
Return value	None
External functs.	None
Description	Applies the function 'f' on each character of the string passed as argument, passing its index as first argument. Each character is passed by address to 'f' to 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 given 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 given 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 given 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 given file descriptor.`

Chapter IV

Bonus part

If you completed the mandatory part, do not hesitate to go further by doing this extra one. It will bring bonus points if passed successfully.

Functions to manipulate memory and strings is very useful. But you will soon discover that manipulating lists is even more useful.

You have to use the following structure to represent a node of your list. Add its declaration to your `libft.h` file:

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

The members of the `t_list` struct are:

- `content`: The data contained in the node.
  `void *` allows to store any kind of data.
- `next`: The address of the next node, or `NULL` if the next node is the last one.

In your Makefile, add a `make bonus` rule to add the bonus functions to your `libft.a`.

The bonus part will only be assessed if the mandatory part is PERFECT. Perfect means the mandatory part has been integrally done and works without malfunctioning. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.

Implement the following functions in order to easily use your lists.

Function name	ft_lstnew
Prototype	t_list *ft_lstnew(void *content);
Turn in files	-
Parameters	content: The content to create the node with.
Return value	The new node
External functs.	malloc
Description	Allocates (with malloc(3)) and returns a new node. The member variable 'content' is initialized with the value of the parameter 'content'. The variable 'next' is initialized to NULL.

Function name	ft_lstadd_front
Prototype	void ft_lstadd_front(t_list **lst, t_list *new);
Turn in files	-
Parameters	lst: The address of a pointer to the first link of a list. new: The address of a pointer to the node to be added to the list.
Return value	None
External functs.	None
Description	Adds the node 'new' at the beginning of the list.

Function name	ft_lstsize
Prototype	int ft_lstsize(t_list *lst);
Turn in files	-
Parameters	lst: The beginning of the list.
Return value	The length of the list
External functs.	None
Description	Counts the number of nodes in a list.

Function name	ft_lstlast
Prototype	t_list *ft_lstlast(t_list *lst);
Turn in files	-
Parameters	lst: The beginning of the list.
Return value	Last node of the list
External functs.	None
Description	Returns the last node of the list.

Function name	ft_lstadd_back
Prototype	void ft_lstadd_back(t_list **lst, t_list *new);
Turn in files	-
Parameters	lst: The address of a pointer to the first link of a list. new: The address of a pointer to the node to be added to the list.
Return value	None
External functs.	None
Description	Adds the node 'new' at the end of the list.

Function name	ft_lstdelone
Prototype	void ft_lstdelone(t_list *lst, void (*del)(void *));
Turn in files	-
Parameters	lst: The node to free. del: The address of the function used to delete the content.
Return value	None
External functs.	free
Description	Takes as a parameter a node and frees the memory of the node's content using the function 'del' given as a parameter and free the node. The memory of 'next' must not be freed.

Function name	ft_lstclear
Prototype	void ft_lstclear(t_list **lst, void (*del)(void *));
Turn in files	-
Parameters	lst: The address of a pointer to a node. del: The address of the function used to delete the content of the node.
Return value	None
External functs.	free
Description	Deletes and frees the given node and every successor of that node, using the function 'del' and free(3). Finally, the pointer to the list must be set to NULL.

Function name	ft_lstiter
Prototype	void ft_lstiter(t_list *lst, void (*f)(void *));
Turn in files	-
Parameters	lst: The address of a pointer to a node. f: The address of the function used to iterate on the list.
Return value	None
External functs.	None
Description	Iterates the list 'lst' and applies the function 'f' on the content of each node.

Function name	ft_lstmap
Prototype	t_list *ft_lstmap(t_list *lst, void *(*f)(void *), void (*del)(void *));
Turn in files	-
Parameters	lst: The address of a pointer to a node. f: The address of the function used to iterate on the list. del: The address of the function used to delete the content of a node if needed.
Return value	The new list. NULL if the allocation fails.
External functs.	malloc, free
Description	Iterates the list 'lst' and applies the function 'f' on the content of each node. Creates a new list resulting of the successive applications of the function 'f'. The 'del' function is used to delete the content of a node if needed.

Chapter V

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your files to ensure they are correct.

Place all your files at the root of your repository.