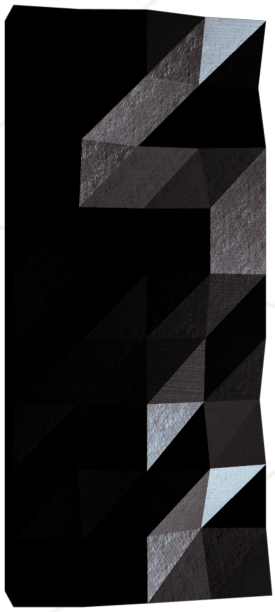

Libft

Your very first own library

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

Version: 18

Contents

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 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

AI Instructions

Context

This project is designed to help you discover the fundamental building blocks of your ICT training.

To properly anchor key knowledge and skills, it's essential to adopt a thoughtful approach to using AI tools and support.

True foundational learning requires genuine intellectual effort — through challenge, repetition, and peer-learning exchanges.

For a more complete overview of our stance on AI — as a learning tool, as part of the ICT curriculum, and as an expectation in the job market — please refer to the dedicated FAQ on the intranet.

Main message

- Build strong foundations without shortcuts.
- Really develop tech & power skills.
- Experience real peer-learning, start learning how to learn and solve new problems.
- The learning journey is more important than the result.
- Learn about the risks associated with AI, and develop effective control practices and countermeasures to avoid common pitfalls.

Learner rules:

- You should apply reasoning to your assigned tasks, especially before turning to AI.

- You should not ask for direct answers to the AI.
- You should learn about 42 global approach on AI.

Phase outcomes:

Within this foundational phase, you will get the following outcomes:

- Get proper tech and coding foundations.
- Know why and how AI can be dangerous during this phase.

Comments and example:

- Yes, we know AI exists — and yes, it can solve your projects. But you're here to learn, not to prove that AI has learned. Don't waste your time (or ours) just to demonstrate that AI can solve the given problem.
- Learning at 42 isn't about knowing the answer — it's about developing the ability to find one. AI gives you the answer directly, but that prevents you from building your own reasoning. And reasoning takes time, effort, and involves failure. The path to success is not supposed to be easy.
- Keep in mind that during exams, AI is not available — no internet, no smartphones, etc. You'll quickly realise if you've relied too heavily on AI in your learning process.
- Peer learning exposes you to different ideas and approaches, improving your interpersonal skills and your ability to think divergently. That's far more valuable than just chatting with a bot. So don't be shy — talk, ask questions, and learn together!
- Yes, AI will be part of the curriculum — both as a learning tool and as a topic in itself. You'll even have the chance to build your own AI software. In order to learn more about our crescendo approach you'll go through in the documentation available on the intranet.

✓ Good practice:

> I'm stuck on a new concept. I ask someone nearby how they approached it. We talk for 10 minutes — and suddenly it clicks. I get it.

✗ Bad practice:

> I secretly use AI, copy some code that looks right. During peer evaluation, I can't explain anything. I fail. During the exam — no AI — I'm stuck again. I fail.

Chapter IV

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.`

IV.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.

IV.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().

Some functions that you must reimplement, such as strlcpy, strlcat, and bzero, are not included by default in the GNU C Library (glibc). To test them against the system standard, you may need to include <bsd/string.h> and compile with the -lbsd flag.
This behaviour is specific to glibc systems. If you are curious, take the opportunity to explore the differences between glibc and BSD libc.

IV.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 concatenating '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.

Chapter V

Bonus part

Once you have completed the mandatory part, consider taking on this extra challenge. Successfully completing this section will earn you bonus points.

Memory and string manipulation functions are 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.
  Using `void *` allows you to store any type 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 in your `libft.a`.

The bonus part will only be evaluated if the mandatory part is perfect. "Perfect" means the mandatory functions are implemented correctly and work without issues. If you fail to meet ALL the mandatory requirements, the bonus part will not be considered 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 store in the new node.`
Return value	A pointer to the new node
External functs.	`malloc`
Description	Allocates memory (using malloc(3)) and returns a new node. The 'content' member variable is initialized with the given 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 node of a list.` `new: The address of a pointer to the node to be added.`
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 the 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 node of a list. new: The address of a pointer to the node to be added.
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 a node as parameter and frees its content using the function 'del'. Free the node itself but does NOT free the next node.

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 all its successors, using the function 'del' and free(3). Finally, set the pointer to the list 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 to apply to each node's content.
Return value	None
External functs.	None
Description	Iterates through the list 'lst' and applies the function 'f' to 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 applied to each node's content. del: The address of the function used to delete a node's content if needed.
Return value	The new list. NULL if the allocation fails.
External functs.	malloc, free
Description	Iterates through the list 'lst', applies the function 'f' to each node's content, and 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 VI

Submission and peer-evaluation

Submit your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Make sure to double-check the names of your files to ensure they are correct.

Place all your files at the root of your repository.

During the evaluation, a brief **modification of the project** may occasionally be requested. This could involve a minor behavior change, a few lines of code to write or rewrite, or an easy-to-add feature.

While this step may **not be applicable to every project**, you must be prepared for it if it is mentioned in the evaluation guidelines.

This step is meant to verify your actual understanding of a specific part of the project. The modification can be performed in any development environment you choose (e.g., your usual setup), and it should be feasible within a few minutes — unless a specific timeframe is defined as part of the evaluation.
You can, for example, be asked to make a small update to a function or script, modify a display, or adjust a data structure to store new information, etc.

The details (scope, target, etc.) will be specified in the **evaluation guidelines** and may vary from one evaluation to another for the same project.

Rnpu cebwrpg va gur 42 Pbzzba Pber pbagnvaf na rapbqrq uvag. Sbe rnpu pvepyr, bayl bar cebwrpg cebivqrf gur pbeerpg uvag arrqrq sbe gur arkg pvepyr. Guvf punyyratr vf vaqvivqhny, jvgu n svany cevmr sbe bar fghqrag. Fgnss zrzoref znl cnegvpvcngr ohg ner abg ryvtvoyr sbe n cevmr. Ner lbh nzbat gur irel svefg gb fbyir n pvepyr? Fraq gur uvagf jvgu rkcynangvbaf gb by@42.se gb or nqqrq gb gur yrnqreobneq. Gur uvag sbe guvf svefg cebwrpg, juvpu znl pbagnva nantenzzrq jbeqf, vf: Jbys bs ntragvir cnegvpyrf gung qvfcebir terral gb lbhe ubzrf qan gung cebjfr lbhe fgbby