**Version: 1.1**

# Libft

## Your very first own library


**Summary**

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


#C #Imperative #Library

# Intellectual Property Disclaimer

All content presented in this training module, including but not limited to texts, images, graphics, and other materials, is protected by intellectual property rights held by Association 42.

## Terms of Use:

- **Personal use:** You are permitted to use the contents of this module solely for personal purpose. Any commercial use, reproduction, distribution, modification, or public display is strictly prohibited without prior written permission from Association 42.
- **Respect for Integrity:** You must not alter, transform, or adapt the content in any way that could harm its integrity.

## Protection of Rights:

Any violation of these terms constitutes an infringement of intellectual property rights and may result in legal action. We reserve the right to take all necessary measures to protect our rights, including but not limited to claims for damages.

*For any questions regarding the use of the content or to obtain authorization, please contact:* `legal@42.fr`

# Contents

# Chapter 1

# 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.

# Chapter 2

# Common Instructions

- Your activity must be written in C.
- Your activity must be written in accordance with the Norm. If you have bonus files or 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.) except for undefined behaviors. If this happens, your activity will be considered non-functional and will receive a 0 during the review.
- All heap-allocated memory space must be properly freed when necessary. No memory leaks will be tolerated.
- If the subject requires it, you must submit a `Makefile` that will compile your source files to the required output with the flags `-Wall`, `-Wextra`, and `-Werror`, using `cc`, and your `Makefile` must not relink.
- Your `Makefile` must at least contain the rules `$(NAME)`, `all`, `clean`, `fclean`, and `re`.
- If your activity allows you to use your `libft`, you must copy its sources and its associated `Makefile` into a `libft` folder. Your activity's `Makefile` must compile the library by using its `Makefile`, then compile the activity.
- We encourage you to create test programs for your activity, even though this work **will not be submitted and will not be graded**. It will give you a chance to easily test your work and your peers' work. You will find these tests especially useful during your defense. Indeed, during the defense, 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 occurs in any section of your work during Deepthought's grading, the review will stop.

# Chapter 3

# AI Instructions

## Context

This activity is designed to help you discover the fundamental building blocks of your 42 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 42 training, 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 activities. 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 4

# Mandatory part

| | |
|---|---|
| **Program Name** | `libft.a` |
| **Files to Submit** | `Makefile, libft.h, ft_*.c` |
| **Makefile** | `NAME, all, clean, fclean, re` |
| **External Functions** | `Detailed below` |
| **Libft authorized** | `n/a` |
| **Description** | `Write your own library: a collection of functions that will be a useful tool for your curriculum.` |

## 4.1 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.

## 4.2 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` pages. 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:

For the character classification functions (`isalpha`, `isdigit`, `isalnum`, `isascii`, `isprint`), the return value must be:

- 1 if the character matches the tested class
- 0 if the character does not match

- `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`

Depending on your current operating system, the calloc man page and the function's behavior may differ. The following instruction supersedes what you can find in the man page: If nmemb or size is 0, then calloc() returns a unique pointer value that can later 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.

## 4.3 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 functions from part 1 can be useful for implementing the functions below.

| **Function Name** | `ft_substr` |
|---|---|
| **Prototype** | `char *ft_substr(char const *s, unsigned int start, size_t len);` |
| **Files to Submit** | - |
| **Parameters** | `s: The string from which to create the substring.`<br>`start: The start index of the substring in the string 's'.`<br>`len: The maximum length of the substring.` |
| **Return Value** | `The substring.`<br>`NULL if the allocation fails.` |
| **External Functions** | `malloc` |
| **Description** | `Allocates (with malloc(3)) and returns a substring from the string 's'.`<br>`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);` |
| **Files to Submit** | - |
| **Parameters** | `s1: The prefix string.`<br>`s2: The suffix string.` |
| **Return Value** | `The new string.`<br>`NULL if the allocation fails.` |
| **External Functions** | `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); |
| Files to Submit | - |
| Parameters | s1: The string to be trimmed.<br>set: The reference set of characters to trim. |
| Return Value | The trimmed string.<br>NULL if the allocation fails. |
| External Functions | 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); |
| Files to Submit | - |
| Parameters | s: The string to be split.<br>c: The delimiter character. |
| Return Value | The array of new strings resulting from the split.<br>NULL if the allocation fails. |
| External Functions | 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); |
| Files to Submit | - |
| Parameters | n: the integer to convert. |
| Return Value | The string representing the integer.<br>NULL if the allocation fails. |
| External Functions | 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)); |
| Files to Submit | - |
| Parameters | s: The string on which to iterate.<br>f: The function to apply to each character. |
| Return Value | The string created from the successive applications of 'f'.<br>Returns NULL if the allocation fails. |
| External Functions | 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 collect the results from the successive applications of 'f'. |

| Function Name | ft_striteri |
|---|---|
| Prototype | void ft_striteri(char *s, void (*f)(unsigned int, char*)); |
| Files to Submit | - |
| Parameters | s: The string on which to iterate.<br>f: The function to apply to each character. |
| Return Value | None |
| External Functions | 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); |
| Files to Submit | - |
| Parameters | c: The character to output.<br>fd: The file descriptor on which to write. |
| Return Value | None |
| External Functions | 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); |
| Files to Submit | - |
| Parameters | s: The string to output.<br>fd: The file descriptor on which to write. |
| Return Value | None |
| External Functions | 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); |
| Files to Submit | - |
| Parameters | s: The string to output.<br>fd: The file descriptor on which to write. |
| Return Value | None |
| External Functions | 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); |
| Files to Submit | - |
| Parameters | n: The integer to output.<br>fd: The file descriptor on which to write. |
| Return Value | None |
| External Functions | write |
| Description | Outputs the integer 'n' to the given file descriptor. |

## 4.4 Part 3 - Linked list

In the third part, you will have to implement functions using a structure in order to manipulate linked lists.

To do so, here is the structure declaration that you should add to your `libft.h` file:

**ft_list.h**

```
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.

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

| **Function Name** | `ft_lstnew` |
|---|---|
| **Prototype** | `t_list *ft_lstnew(void *content);` |
| **Files to Submit** | - |
| **Parameters** | `content: The content to store in the new node.` |
| **Return Value** | `A pointer to the new node` |
| **External Functions** | `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);` |
| **Files to Submit** | - |
| **Parameters** | `lst: The address of a pointer to the first node of a list.`<br>`new: The address of a pointer to the node to be added.` |
| **Return Value** | None |
| **External Functions** | None |
| **Description** | `Adds the node 'new' at the beginning of the list.` |

| Function Name | ft_lstsize |
|---|---|
| Prototype | int ft_lstsize(t_list *lst); |
| Files to Submit | - |
| Parameters | lst: The beginning of the list. |
| Return Value | The length of the list |
| External Functions | None |
| Description | Counts the number of nodes in the list. |

| Function Name | ft_lstlast |
|---|---|
| Prototype | t_list *ft_lstlast(t_list *lst); |
| Files to Submit | - |
| Parameters | lst: The beginning of the list. |
| Return Value | Last node of the list |
| External Functions | 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); |
| Files to Submit | - |
| Parameters | lst: The address of a pointer to the first node of a list.<br>new: The address of a pointer to the node to be added. |
| Return Value | None |
| External Functions | 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 *)); |
| Files to Submit | - |
| Parameters | lst: The node to free.<br>del: The address of the function used to delete the content. |
| Return Value | None |
| External Functions | free |
| Description | Takes a node as parameter and frees its content using the function 'del'. Frees the node itself but does NOT free the next node. |

| Function Name | ft_lstclear |
|---|---|
| Prototype | void ft_lstclear(t_list **lst, void (*del)(void *)); |
| Files to Submit | - |
| Parameters | lst: The address of a pointer to a node.<br>del: The address of the function used to delete the content of the node. |
| Return Value | None |
| External Functions | 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 *)); |
| Files to Submit | - |
| Parameters | lst: The address of a pointer to a node.<br>f: The address of the function to apply to each node's content. |
| Return Value | None |
| External Functions | 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 *)); |
| Files to Submit | - |
| Parameters | lst: The address of a pointer to a node.<br>f: The address of the function applied to each node's content.<br>del: The address of the function used to delete a node's content if needed. |
| Return Value | The new list.<br>NULL if the allocation fails. |
| External Functions | 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 5

# Readme Requirements

A README.md file must be provided at the root of your Git repository. Its purpose is to allow anyone unfamiliar with the activity (peers, staff, recruiters, etc.) to quickly understand what the activity is about, how to run it, and where to find more information on the topic.
The README.md must include at least:

- The very first line must be italicized and read: *This activity has been created as part of the 42 curriculum by <login1>[, <login2>[, <login3>[...]]].*
- A "**Description**" section that clearly presents the activity, including its goal and a brief overview.
- An "**Instructions**" section containing any relevant information about compilation, installation, and/or execution.
- A "**Resources**" section listing classic references related to the topic (documentation, articles, tutorials, etc.), as well as a description of how AI was used — specifying for which tasks and which parts of the activity.

➠ **Additional sections may be required depending on the activity** (e.g., usage examples, feature list, technical choices, etc.).

Any required additions will be explicitly listed below.

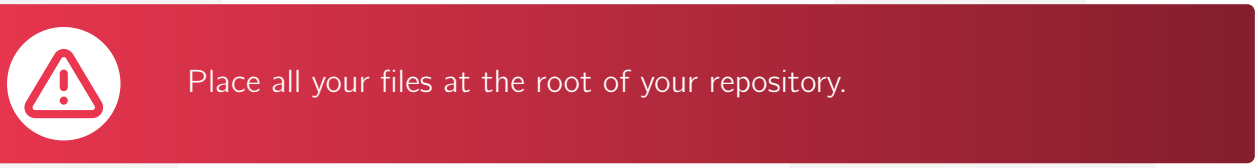
- A detailed description of the library created for this project must also be included.

English is recommended; alternatively, you may use the main language of your campus.

# Chapter 6

# 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 peer-evaluation. Don't hesitate to double check the names of your files to ensure they are correct.




During the review, a brief **modification of the activity** 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 activity**, you must be prepared for it if it is mentioned in the review guidelines.

This step is meant to verify your actual understanding of a specific part of the activity. 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 review.
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 **review guidelines** and may vary from one review to another for the same activity.