Version: 1.0

Selection

C-Libft

Summary

Create a static C library (libft.a) and automate its compilation using a Makefile.

#C #Library #Makefile

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

Instructions

- Only this page will serve as a reference: do not trust rumors.
- Watch out! This document could potentially change up until submission.
- Make sure you have the appropriate permissions on your files and directories.
- You have to follow the submission procedures for all your exercises.
- Your exercises will be checked and graded by your fellow classmates.
- Additionally, your exercises will be checked and graded by a program called Moulinette.
- Moulinette is very meticulous and strict in its evaluation of your work. It is entirely automated, and there is no way to negotiate with it. So, to avoid bad surprises, be as thorough as possible.
- Moulinette is not very open-minded. It won't try to understand your code if it doesn't adhere to the Norm. Moulinette relies on a program called `norminette` to check if your files respect the norm. TL;DR: it would be foolish to submit work that doesn't pass `norminette`'s check.
- These exercises are carefully laid out by order of difficulty - from easiest to hardest. We `will not` consider a successfully completed harder exercise if an easier one is not perfectly functional.
- Using a forbidden function is considered cheating. Cheaters get -42, and this grade is non-negotiable.
- You'll only have to submit a main() function if we ask for a program.
- Moulinette compiles with these flags: -Wall -Wextra -Werror, and uses `cc`.
- If your program doesn't compile, you'll get `0`.
- You cannot leave any additional files in your directory other than those specified in the subject.
- Got a question? Ask your peer on the right. Otherwise, try your peer on the left.
- Your reference guide is called `Google / man / the Internet / ...`.
- Check out the Slack Piscine.

- Examine the examples thoroughly. They could very well call for details that are not explicitly mentioned in the subject...
- By Odin, by Thor! Use your brain!!!

Do not forget to add the *standard 42 header* in each of your .c/.h files. Norminette checks its existence anyway!

Norminette must be launched with the *-R CheckForbiddenSourceHeader* flag. Moulinette will use it too.

Context

The C Piscine is intense. It's your first big challenge at 42 — a deep dive into problem-solving, autonomy, and community.

During this phase, your main objective is to build your foundation — through struggle, repetition, and especially **peer-learning** exchange.

In the AI era, shortcuts are easy to find. However, it's important to consider whether your AI usage is truly helping you grow — or simply getting in the way of developing real skills.

The Piscine is also a human experience — and for now, nothing can replace that. Not even AI.

For a more complete overview of our stance on AI — as a learning tool, as part of the ICT curriculum, and as a growing expectation in the job market — please refer to the dedicated FAQ available 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 2

Foreword

42-Cloves Garlic Chicken Recipe

- **Ingredients:**
  - 1 whole chicken, cut into 8 pieces
  - 42 cloves of garlic (yes, 42!)
  - 1/2 cup olive oil
  - 1 lemon, juiced
  - 1 tbsp dried thyme
  - 1 tbsp dried rosemary
  - Salt and pepper to taste
- **Instructions:**

1. Preheat the oven to 400°F (200°C).
2. In a large bowl, combine olive oil, lemon juice, thyme, rosemary, salt, and pepper.
3. Add the chicken pieces to the bowl and toss to coat evenly.
4. Place the chicken in a baking dish and tuck the 42 garlic cloves around the chicken.
5. Drizzle any remaining marinade over the chicken.
6. Bake for 45-50 minutes, or until the chicken is golden brown and cooked through.
7. Serve hot, with crusty bread to spread the softened garlic.

Chapter 3

Tutorial

- As you will now start using your own library you just created, including more and more `.c files`, compiling everything from zero to get your fully operational software will take more and more time and commands to execute. As we are programmers, let's use a program to make the fastidious work for us.

- The program we will use is **make**. It comes with a set of features that will allow us to easily do all the needed tasks from source code to the final product we target. The **make** program, that is part of the system, expects to find instructions in a specific file called **Makefile**. Simply typing `make` in the shell should do all the needed actions to create your entire software.

- Today, you'll create your very first **Makefile**, understand its syntax, and discover a very small subset of features to achieve our current goals. Up to you in the future to reveal and use more complex or advanced features of **make** and **Makefile**.

  Using a **Makefile** implies that you are using a compiling feature, called "separate compilation". You'll find plenty of explanations out there. Basically, instead of directly creating your piece of software directly out of the **.c** files, you'll make a pause at an intermediate step: the object files (**.o** files). It's already compiled code, but only for the functions present in the **.c** file. You can't use it, except assembling it with other object files to create your entire software. Just like the pieces of a puzzle. The main advantage of separate compilation is to avoid re-compile a **.c** file that has not been modified since the last compilation. With HUGE projects (like compiling google chromium that can take a lot of time) it's definitely necessary.

- Create a working directory for this tutorial and place your `ft_strdup.c` file and a `main.c` file that tests this function. If you don't have these files, create them now. Make sure they compile and work properly with manual compilation before continuing.

- In this same directory, create a file named exactly `Makefile` (watch the case, no extension). Open it with your text editor.

- Now let's write our first **Makefile**. A simple and classic **Makefile** is structured in two parts: the declarations and the rules. The declarations are where you define several elements, just like shell environment variables.

Syntax is: **VARNAME = string**. Usually, the variable name on the left is in uppercase. If you aren't sure, check with your neighbors.

Start by typing these lines in your Makefile:

Makefile

```
NAME = test_strdup
SOURCES = ft_strdup.c main.c
OBJECTS = $(SOURCES:.c=.o)
```

This part correspond to the declarations.

- Let's add a rule. Add the following in your **Makefile** :

Makefile

```
$(NAME): $(OBJECTS)
	cc -o $(NAME) $(OBJECTS)
```

By default, when typing **make**, only the very first rule will be executed.

- **make** comes with implicit rules. The most used one compiles any **.c** file into a **.o** file automatically. Save your `Makefile`, then type `make` in your terminal and observe what happens.
- Type `ls` to see the files created. You should see **.o** files and your executable. Run your executable to test it works.
- Add a rule **clean** that removes the all the .o files.
- Type `make clean` then `ls` to verify the **.o** files have been removed.
- Add a rule **fclean** that calls **clean** rule, then deletes the program.
- Type `make clean` then **list** your directory. Everything should be cleaned up. Recompile with `make` to verify everything still works.
- Add **compilation flags** by adding this line after your variables:

Makefile

```
CFLAGS += -Wall
```

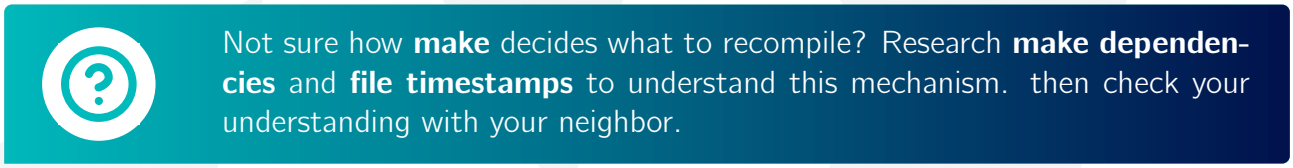
Clean everything with `make fclean` then recompile with `make`. Observe how the -Wall option appears automatically in the compilation.

- Create a subdirectory called `ft_range` and place your `ft_range.c` and a `main.c` inside. Now modify your `Makefile` to handle both functions.

  Simply type `make` to compile both `ft_strdup` and `ft_range`. You can use subdirectories. Test both executables to verify they work.

- Experiment with **make**'s behavior. Delete one **.o** file with `rm ft_strdup.o`, then type `make all`. Observe which files get recompiled and which don't.

- Type `make all` a second time without changing anything. What happens and why?



- Add the **-Wall** option to the compilation of each **.c**

- Make sure that rules **clean** and **fclean** applies to both `ft_strdup` and `ft_range`.

Chapter 4

Exercise 0: libft

	Exercise: 0	
libft		
Directory: *ex0/*		
Files to Submit: `libft_creator.sh`, `ft_putchar.c`, `ft_swap.c`, `ft_putstr.c`, `ft_strlen.c`, `ft_strcmp.c`		
Authorized: `write`		

- Create your `ft` library. It'll be called `libft.a`.
- A shell script called `libft_creator.sh` will compile source files appropriately and will create your library.
- This library should contain all of the following functions :

Declaration

```
void    ft_putchar(char c);
void    ft_swap(int *a, int *b);
void    ft_putstr(char *str);
int     ft_strlen(char *str);
int     ft_strcmp(char *s1, char *s2);
```

- We'll launch the following command-line :

Terminal:

```
?> sh libft_creator.sh
```

Chapter 5

Exercise 1: Makefile

	Exercise: 1	
Makefile		
Directory: *ex1/*		
Files to Submit: `Makefile`		
Authorized: `None`		

- Create the `Makefile` that'll compile a library `libft.a`.
- Your makefile should print all the command it's running.
- Your makefile should not run any unnecessary command.
- The `Makefile` will get its source files from the "srcs" directory.
- Those files will be: ft_putchar.c, ft_swap.c, ft_putstr.c, ft_strlen.c, ft_strcmp.c
- The `Makefile` will get its header files from the "includes" directory.
- Those files will be: ft.h
- It should compile the .c files with cc and with -Wall -Wextra -Werror flags in that order.
- The lib should be at the root of the exercise.
- .o files should be near their .c file.
- The `Makefile` should also implement the following rules: `clean`, `fclean`, `re`, `all` and of course `libft.a`.
- A `main.c` file at the root should also be compiled with the same flags. Its .o file should be included in the library.
- Running just `make` should be equal to `make all`
- The rule `all` should be equal to `make libft.a`.
- The rule `clean` should remove all the temporary generated files.

- The rule `fclean` should be like a `make clean` plus all the binary made with `make all`.
- The rule `re` should be like a `make fclean` followed by `make all`.
- Your makefile should not compile any file for nothing.
- We'll only fetch your Makefile and test it with our files.

Watch out for wildcards!

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 defense. Don't hesitate to double check the names of your files to ensure they are correct.

You need to return only the files requested by the subject of this project.