Makefiles

Introduction

A Makefile is a file, read by the `Make` program, which executes all the commands and rules in the Makefile.

Remember

Remember the first two days of C Pool. You were supposed to use bash and build your own aliases and scripts. `Make` is kind of a custom build script that makes your programmer-life really easier.

At EPITECH, you'll be using Makefiles to compile your code, using a `compiler`.

Generic Makefile

A simple Makefile is composed of your project source files (the .c files) and some `rules` to make your `Make` command work.

You have to list your source files like this:

```
SRC = ./main.c \
      ./file.c
```

After that, you can use them to build your objects. It will take all `.c` files in `$(SRC)` and compile them into `.o` files.

```
OBJ = $(SRC:.c=.o)
```

For the compilation there is a feature that allow you to compile each `.c` with flags, it's the `+=`. For example let's add verification of errors flags : `-Werror -Wextra` and a flags to find `.h` of your project : `-I./include`. You can call this variable `CFLAGS` for compilation's flags.

```
CFLAGS += -Werror -Wextra -I./include
```

Be careful !

You don't have to call this variable in your Makefile, he will solo add it to the compilation of your `.c`.

Now, set the name of your final binary using `NAME`, so the AutoGrader can find your binary correctly.

```
NAME = binary_name
```

Then, it is mandatory to create a $(NAME) rule that will execute other rules, and render a binary.

```
$(NAME): $(OBJ)
	gcc -o $(NAME) $(OBJ)

all:	$(NAME)
```

Pro-tip

When you have a rule like `$(NAME)`, the rules put in the same line will be used as mandatory rules. After those rules have been called, the command on the next lines will be called.

For instance, this will execute the `ls` command without executing any previous rule.

```
list:
	ls
```

You can also have some rules that permit you to clean-up your folder without having to do it manually.

For instance, this `clean` will remove `.o` files. Also, `fclean` will remove `.o` files and the binary. `re` will do `fclean` and re-make your binary.

```
clean:
	rm -f $(OBJ)

fclean: clean
	rm -f $(NAME)

re:	fclean all
```

Don't forget to put a `.PHONY`, in order to avoid relinking. Put all the rules you use.

```
.PHONY: all clean fclean re
```

And that's pretty much it ! Your Makefile is now ready to use.

Criterion Makefile

At EPITECH, you use `criterion` for unit tests. In order to make it clean there is a approach given by EPITECH.

First of all, you have to add a new rule to your main Makefile, according to EPITECH this rule should be named `tests_run`.

Pro tip

In order to make it cleaner we recommend you to another Makefile in the tests directory and link to the main. To call a Makefile rule of your tests Makefile just type : make -C tests/ [rule_name] in your main Makefile.

The `tests_run` rule should compile your sources files `.c` and your tests files. This rule must launch your binary `./unit-tests`.

Mendatory !

You never have to put your main function in the source files that you compile for unit tests : Criterion have his own.

Your tests must compile with the CFLAG `--coverage` (see Generic Makefile). This flag will create `.gcda` and `.gcno` of your sources files.

Tip

Make a rule to clean all your `.gcda` and `.gcno` files.

Now, when you launch your tests_run rule, your binary of tests should compile again and execute so that you can see if you passed tough your tests. You should see your files from –coverage. You can use the gcov [files] to see how many line were executed when you launch your unit tests.

Clear all `.gcda`, `.gcno` and `.c.gcov` and you can push it to the AutoGrader !

Library Makefile

Advanced Makefile