Piscine — Tutorial D4

version **#v3.2.0**

Assistant C/UNIX 2021 <assistants@tickets.assistants.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Contents

*https://intra.assistants.epita.fr

1 Memory by allocation

1.1 Introduction

Up until now, you've been using a fixed amount of memory when writing C programs. You've been using variables that only existed during a function call, or a specific amount of memory that was used during the whole lifetime of your program.

There will be a lot of cases where you cannot plan ahead for the amount of memory your program will need. In order to solve this problem, you can manage some amounts of memory during the execution of your program using the `malloc(3)` and `free(3)` functions.

1.2 Dynamic memory

In C, management of the dynamically allocated memory space is **manual**: allocations and deallocations are **explicitly** managed by you, the developer. Memory allocated by a call to `malloc(3)` is **not** automatically deallocated at the end of the function.

Your allocator is a set of functions which manage all the memory chunks you allocate and free to fit them into a large memory region, owned by the allocator, and used to store all these blocks. You just need to request a memory region of a desired size to your allocator, and it will return a pointer to an area of such size if it is available.

1.3 Memory allocation

The following block of code is an example of using C's standard memory allocator: `malloc(3)` defined in `stdlib.h`.

```c
#include <stdlib.h>

int *toto(void)
{
    return malloc(sizeof(int));
}
```

As you can see, `malloc(3)` returns a pointer. Here we have a function `toto` returning a pointer to an area large enough to hold an `int` value. The `sizeof` keyword is used to determine the memory size required for the `int` type. We give this size to `malloc(3)` and it will return a chunk of dedicated space in which you can store your `int` element.

According to the prototype of the `malloc(3)` function, the type returned is `void *`, this represents a generic pointer to unknown data. It's up to you to assign this pointer to a pointer of a specific type; which allows your compiler to ensure that you are correctly using your pointer. For example, if you specify a `char` pointer where an int pointer is expected, an error can be detected. This error would not have occurred had you been using a generic pointer. That's why the function `toto` returns a pointer to an integer (`int *`).

If the memory is full, the `malloc(3)` function will return `NULL`. This behavior is described in `malloc(3)`'s man page. `NULL` is a special value, it usually points to the first address in your program's memory. Why is that so? In the range of memory available for your program, some parts are not accessible. This

is the case for the beginning of the virtual address space of your program. The first address is not accessible, and if you try to access it, a segmentation fault will occur. This behaviour will allow you to use the zero address represented by `NULL` as an invalid address for invalid pointers. Thus, if you try to access it, your program will crash. This address is guaranteed to be invalid (except if a kernel programmer makes a joke, but that's not funny), so it can be used by `malloc(3)` to notify you that the allocation cannot be done.

Be careful!

Always[1] check `malloc(3)`'s return value! If the allocation fails and returns `NULL`, your program will crash when it will use the incorrectly allocated pointer (This may happen later in your code, making debugging harder ...). Of course, your ACUs may give your program a `malloc(3)` that will return `NULL` during tests...

[1] **Always**

We **strongly** advise you to always use `NULL` (defined in the header `stddef.h`, but included in `stdlib.h`) to initialize your pointers. Ideally, a pointer must contain either a valid address or `NULL`. Never leave an uninitialized pointer, because the address it holds can be anything, and that address may be `NULL` but may also be another an invalid one, or worse, a valid address somewhere else in memory.

1.4 Memory deallocation

As said previously, memory areas allocated by `malloc(3)` are not destroyed (freed or unallocated) automatically. We need a function to deallocate the memory's areas at the addresses returned by `malloc(3)`. This function is named `free(3)` and takes a pointer to the memory that must be released as a parameter.

Whenever you don't need the memory allocated by `malloc(3)` anymore, you should free it using `free(3)`.

Forgetting to do so can cause what are called *memory leaks*. Those are some of the worst mistakes that can occur in a program. If a program with *memory leaks* runs for a long period of time (a server for example), it will completely fill the RAM and will slow the system down, or can even cause it to shutdown. *Memory leaks* are also some of the hardest bugs to find. You should **always**[2] keep in mind where you will free allocated memory.

Once you call `free(3)`, your pointer still holds the address it did before the call, which is not valid anymore. Dereferencing this address leads to an undefined behavior. If your pointer variable still exists after your `free(3)` (not right before function's end), you should assign it to `NULL` to avoid confusion.

For example:

```
int *i_ptr = NULL;

/* The memory space that i_ptr points to is allocated */
i_ptr = malloc(sizeof(int));

if (!i_ptr) /* malloc returned NULL, this pointer is not valid */
    return /* whatever */;
```

(continues on next page)

[2] **Always**.

(continued from previous page)

```c
*i_ptr = 42; /* let's fill this memory with a value */

int b = *i_ptr; /* b's value is 42 */

free(i_ptr); /* memory chunk is given back */

i_ptr = NULL; /* mark this pointer as NULL to avoid keeping an invalid address */

/* Do some stuff and return */
```

Be careful not to mistake a pointer and the memory's area to which it points! In the previous example, the pointer (i.e. the variable `i_ptr`) is allocated automatically on the stack, and the area pointed by `i_ptr` is allocated manually with `malloc(3)`.

The man page of function `free(3)` specifies that it takes as parameter any pointer returned by `malloc(3)`, thus giving `NULL` pointer to `free(3)` is valid (but won't do anything).

1.4.1 Exercises: memory

addresses

Create variables on the stack, and display their address using `%p` of `printf(3)`.

Do the same using addresses returned by `malloc(3)`.

Explain the difference between two values.

create_array

You want to create arrays of `int`, but with a size only known at runtime.

```c
int *create_array(size_t size);
```

Return a pointer to a memory region containing `size` integers (`int`). Return `NULL` if you cannot allocate the memory.

Tips

`size_t` is available in `stddef.h`, which is included by `stdlib.h`.

free_array

You do not need the previously allocated array anymore.

```
void free_array(int *arr);
```

Free the memory used by the given array. Do not do anything if `arr` is `NULL`.

read_and_inc

```
void read_and_inc(int *v);
```

Using `printf`, display the integer pointed by `v` and increment it by one.

my_strdup

```
char *my_strdup(const char *str);
```

Allocate a char array large enough to hold the `str` string (null terminated) and copy the value of `str` into this memory area. Return the pointer to this memory. We must be able to pass the returned pointer to `free(3)`. Return `NULL` if the allocation failed.

Use `printf(3)` to display the string returned by `my_strdup` to check consistency.

my_strndup

```
char *my_strndup(const char *str, size_t n);
```

Same as `my_strdup` but copies at most `n` bytes. If `str` is shorter than `n`, acts as `my_strdup`. `n` does not include the terminating null byte. The pointer returned should be freed using `free(3)`. Returns `NULL` if the allocation failed.

Use `printf(3)` to display the string returned by `my_strndup` to check consistency.

2 Makefiles

2.1 Context

For now, in order to compile your program you did something like:

```
42sh$ gcc -std=c99 -Wall -Wextra -Werror -pedantic hello_world.c -o hello_world
```

We can all agree that this is inconvenient and fastidious to type[1].

[1] And don't talk to us about your .bash_history, it is equally fastidious to search the command line you want.

When working on actual projects, you will also have more than one source file, and quickly, your simple one line command can become very long and if you want to move a file it will be complex to edit[2].

And another problem that you will see soon enough: compiling a big project can take a lot of time and be boring[3], you don't want to recompile the whole project each time you edit a file.

So the problems are:

- We don't want to type a long command
- We want to be able to edit our compilation command easily
- We don't want to recompile our project each time we edit a file
- And we want other people to easily compile our project

2.2 What is make ?

In response to those issues, the POSIX standard contains a tool used to handle project builds: `make`. The implementation of `make` that we will use this semester is `GNU make`[4] which was released by the FSF as part of the GNU project. `GNU make` implements all the features defined in the POSIX standard for `make` and has some nice extensions as well.

`make`, is a tool used to simplify the task of building programs composed of many source files. It can be configured through a file named `Makefile`, `makefile`, or `GNUmakefile`. `make` will first search for a `GNUmakefile`, then `makefile`, and finally `Makefile` if it didn't find the previous ones. However, it is recommended in the official GNU/Make documentation to call your makefile `Makefile`, and we expect you to follow this convention for your submissions.

`make` parses rules and variables from the `Makefile` which are mainly used to build the project. `make` will only re-build things that need to be re-built by comparing modification dates of targets with their dependencies.

A Makefile typically starts with a few variable definitions, followed by a set of target entries. Each variable from the Makefile is used with `$(VARIABLE)` and can be declared at the top of the file as:

```
VARIABLE = VALUE
```

Finally, each rule follows the convention:

```
target: dependency1 dependency2 ...
    command
    ...
```

A target is usually the name of the file generated by the Makefile rule. The most common examples of targets are executables or object files. It can also be the name of an action to carry out, for example `all`, `clean`, ...

A **dependency** (or "prerequisite") is a file that needs to exist in order to create the **target**. A **target** can also be a **dependency** for another **target**. When evaluating a rule, `make` will analyze its dependencies and if one of them is a target of another rule, `make` will evaluate it too before the one it **depends** on.

[2] `fc(1)` is out of the discussion as well.
[3] https://xkcd.com/303/
[4] https://www.gnu.org/software/make/manual/

A command (or "recipe") is an action that `make` carries out. Be careful, each command is **preceded by a tab** (`\t`), otherwise, your Makefile will not work.

2.3 Invocation

2.3.1 Basics

Let's start easy. First, you need a `hello_world.c` like the following one:

```c
#include <stdio.h>

int main(void)
{
    puts("Hello World !");
    return 0;
}
```

And a `Makefile`:

```
CC=gcc
CFLAGS=-std=c99 -Wall -Wextra -Werror -pedantic

hello_world: hello_world.o
	$(CC) -o hello_world hello_world.o

hello_world.o: hello_world.c
	$(CC) $(CFLAGS) -c -o hello_world.o hello_world.c
```

Be careful!

Remember that the `command` part of a `rule` **must** start with a tabulation.

Now try:

```
42sh$ make hello_world
gcc -std=c99 -Wall -Wextra -Werror -pedantic -c -o hello_world.o hello_world.c
gcc -o hello_world hello_world.o
```

So what is going on here ?

First you have two `variables`:

- `CC` that means C Compiler
- `CFLAGS` this one is explicit, it concerns the.. C Flags

Then you have two `rules`. Those are the core of a `Makefile`. Let's look at the first one.

First we want to produce a binary called `hello_world`, this is our **target**.

In order to build our binary, we need the object `hello_world.o`, we say that our **target** `hello_world` depends on the existence of the object `hello_world.o`. Thus `hello_world.o` is a **dependency** of the **target** `hello_world`.

Now that our `rule` specifies the **target** it produces and what it depends on, we need to specify how it should be produced. Remember our `variables` ? We will need them now. In order to `expand`[5] a Makefile `variable`, you need to put it between parentheses[6] with a dollar before it.

So if we summarize this rule, it will produce the **target** `hello_world` that depends on the object `hello_world.o` using the following command line:

```
gcc -o hello_world hello_world.o
```

But the object `hello_world.o` does not exist! That's why we have the second rule! If a **dependency** does not exist, `make` will search for a rule that can produce it and execute it before the first.

The second rule follows the same principle. You should be able to understand what it does by your own.

Going further...

The default `rule` called when typing `make` is the first one of the file. Thus, here, typing `make` or `make hello_world` will have exactly the same behavior.

If you run `make` again without updating any file, you will have a message like this:

```
42sh$ make hello_world
make: 'hello_world' is up to date.
```

If a target already exists, `make` will rebuild it if its dependencies don't exist or if they have been updated since the last build.

Going further...

You can force `make` to rebuild all targets with the option `-B`.

If you modify the `hello_world.c` and retry `make`, it will rebuild everything.

Another interesting thing to note is `make`'s `-n` option which asks `make` to print the commands it would have run instead of running them.

```
42sh$ cat Makefile
all:
    echo toto
42sh$ make
echo toto
toto
42sh$ make -n
echo toto
```

[5] The `expansion` is the concept of replacing a variable by its value before interpreting the line.
[6] You might see braces instead of parentheses, they work the same. It is as you prefer.

2.3.2 Implicit rules

There are some `implicit rules` which are already defined by `make`. As you can see, `implicit rules` are called whenever `make` tries to build a target for which you did not explicitly define a rule. If you refrain from specifying recipes or rules, then `make` will look for which `implicit rule` to use.

For example, in order to produce a `.o`, you will almost always call this command line:

```
gcc -std=c99 -Wall -Wextra -Werror -pedantic -c -o hello_world.o hello_world.c
```

Where the wanted `.o` depends on the corresponding `.c`[7].

Considering that, if you want to produce a `hello_world.o` `make` will automatically search for the corresponding `hello_world.c` and call:

```
$(CC) $(CFLAGS) -c -o hello_world.o hello_world.c
```

Similarly, if a **target** does not have any extension, it most likely is a binary. Thus, `make` knows that the command to produce it depends on the corresponding `.o`. And call the following command:

```
$(CC) -o hello_world hello_world.o
```

Considering this, our Makefile now will look like this:

```
CC = gcc
CFLAGS = -std=c99 -pedantic -Wextra -Wall -Werror

all: hello_world
```

Here, the target `all` depends on the target `hello_world` and does nothing, so `make all` will execute the rule that produces `hello_world` and do nothing.

In other word, because you did not mention a rule nor a recipe for the dependency `hello_world` nor `hello_world.o`, `make` will update the `hello_world.o` and `hello_world` files itself, even if they don't exist.

```
42sh$ make
gcc -std=c99 -Wall -Wextra -Werror -pedantic -c -o hello_world.o hello_world.c
gcc -o hello_world hello_world.o
```

Produces the exact same output as before!

Tips

Note that `make` was called with no argument. In this case, the first rule defined is called. By convention, it is almost always called `all` and has all the main rules (except cleaning ones) of the Makefile as prerequisites.

[7] If you want to produce a `gell.o` from a `foo.c` you are a strange person.

2.3.3 Finally

Now that you have seen how a Makefile works, we provide you with a simple yet functional Makefile in order to help you during your work.

```
CC = gcc
CFLAGS = -std=c99 -Wall -Wextra -Werror -pedantic

OBJS = hello_world.o
BIN = hello_world

all: $(BIN)

$(BIN): $(OBJS)

clean:
	$(RM) $(BIN) $(OBJS)
```

Notice how we put our targets in variables, it makes it simple to maintain and edit.

We specify that `$(BIN)` depends on `$(OBJS)`. If you have several files, it will tell `make` which object files to use (and build) in order to compile your binary.

`make clean` will remove the executable and the object files. Notice how we used the variable `RM` without defining it. `make` already defines some variables that we can use or redefine. `RM` as the value `rm -f` by default, `CC` has the value `cc`[8], ...

```
42sh$ make
gcc -std=c99 -pedantic -Wextra -Wall -Werror   -c -o hello_world.o hello_world.c
gcc   hello_world.o   -o hello_world
42sh$ make clean
rm -f hello_world hello_world.o
```

3 Makefiles advanced

3.1 Reminders

`make` allows you to automate the build of a program through a set of rules defined in a `Makefile`. A rule specifies a *target*, that may depend on other files or rules as *prerequisites*, and the *commands* to execute when the rule is called. It has the following syntax:

```
target: prerequesites ...
    command
    ...
```

[8] Which is a symbolic link to a c compiler.

3.2 Setting variables

There are different ways to set variables in a Makefile.

3.2.1 Lazy set

```
VARIABLE = VALUE
```

This is the usual way to set variables, values within it are recursively expanded when the variable is used. For example:

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:
    echo $(foo)
```

The previous `Makefile` will result in:

```
42sh$ make all
echo Huh?
Huh?
```

3.2.2 Set if not set

Another way to set variables is:

```
VARIABLE ?= VALUE
```

Basically, the variable will be set to the value only if it has not already been set before.

Note that the variable's previous declaration can be in the Makefile, as well as the environment.

This kind of assignment can be particularly useful to specify names of external tools. For example, if someone does not have `gcc` but only `clang`, it allows him to specify his favourite compiler in his environment without editing the makefile.

```
42sh$ cat -e Makefile
CC ?= gcc$
42sh$ export CC=clang
42sh$ make main
clang main.c -o main
```

```
42sh$ cat -e Makefile
CC = gcc$
42sh$ export CC=clang
42sh$ make main
gcc main.c -o main
make: gcc: Command not found
make: *** [main] Error 127
```

As you can see, with the lazy set, `make` produces an error.

3.2.3 Appending

It can also be useful to add more text to a previously defined variable. You can do it with:

```
VARIABLE += VALUE
```

When the variable has not been defined before, += acts just like =.

3.3 Value assignment

There are three ways available to the user to assign a variable its value for `make`:

- In an assignment in the Makefile, as we just showed
- In the environment
- During `make`'s invocation. It is an *overriding value*

3.3.1 During invocation

```
42sh$ make -n main CC=clang
clang -std=c99 -pedantic -Wextra -Wall -Werror    main.c   -o main
```

Here the CC value given in command line overrides the one defined previously in the Makefile.

3.3.2 In the environment

```
42sh$ CC=clang make -n main
clang -std=c99 -pedantic -Wextra -Wall -Werror    main.c   -o main
```

```
42sh$ export CC=clang
42sh$ make -n main
clang -std=c99 -pedantic -Wextra -Wall -Werror    main.c   -o main
```

As we saw earlier, these examples work only if CC is not set or if it is set with `?=` in the makefile.

3.4 Automatic variables

Make embeds a few special variables that are automatically defined:

- `$@`: gets the name of the Makefile rule
- `$^`: gets all the prerequisites
- `$<`: gets the first prerequisite

These 3 are probably the most useful but you can find the other ones in the GNU/Make official documentation: https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

3.5 .PHONY

Now that you know how Makefile rules work, you may start to see a problem. What if we create a rule called clean that cleans the directory, but there is a file called clean in the current directory? How will make choose? Let's try it!

```
42sh$ cat Makefile
CC ?= gcc
CFLAGS = -std=c99 -Wall -Wextra -Werror -pedantic

all: main

clean:
        $(RM) *.o main
42sh$ make clean
rm -f *.o main
42sh$ touch clean
42sh$ make clean
make: `clean' is up to date.
```

As you can see, the rule clean works as long as the file clean does not exist. But once the file clean exists, make sees that the file is up to date and hence ignores the rule.

You would have the same problem if you created a file called all, because this problem occurs with all the rules whose target is not a file name.

There is a simple way to avoid this problem. You just have to define a .PHONY rule that takes all the rules that do not represent file names as dependencies.

```
42sh$ cat Makefile
CC ?= gcc
CFLAGS = -std=c99 -Wall -Extra -Werror -pedantic

.PHONY: all clean

all: main

clean:
        ${RM} *.o main
42sh$ make clean
rm -f *.o main
42sh$ touch clean
42sh$ make clean
rm -f *.o main
```

As you can see, even when the clean file exists, make behaves as expected.

We strongly encourage you to check out Marwan's GNU-Make tutorial[1].

[1] https://slashvar.github.io/2017/02/13/using-gnu-make.html

4 Toolchain

4.1 Introduction

A computer is a basic automaton: the user provides it with a series of instructions that will run sequentially, one after another. Its great strength is especially to do these tasks very quickly.

Unfortunately, a transistor still being only a transistor, the computer is also very stupid. The instructions that it can execute remain extremely basic: add this number to another, read this memory cell, etc. It is only by combining such instructions in large numbers that we obtain a program that makes sense and is useful to the end user. This set of instructions is expressed in **binary code**: a sequence of 0 and 1 which is (almost) incomprehensible to a human being. It is therefore unthinkable to hope about creating a slightly advanced program by manually writing such binary code.

To simplify the programmer's job, the **assembly language** was created. The assembly language is just a bijection between binary code and a readable language. There is a utility tool called **assembler**, which translates such assembly code into machine-readable binary code.

This programming method remains very restrictive: the programmer still needs to write hundreds of lines of code to achieve the most basic tasks. And, as you will soon discover, more code means more bugs. Besides, assembly code is closely linked to the architecture we want it to be executed on (its target): for example, an assembly code for a PC (x86-64 for example) is not compatible with your phone (ARM architecture) or any device or PC with a different architecture.

It was therefore required to talk to computers using higher level and more advanced programming languages in order to simplify the programmer's job and increase portability. Such languages usually come with a utility, called **compiler**, that will translate the language into assembly for the target platform.

4.2 Overview

Compiling a program is actually a misnomer: the compilation is only one step in the creation of binary files.

The creation of a binary from C source files is actually done in four main steps that each produce files easily identifiable by their extension: it's called the *toolchain*! The steps of this chain are:

Preprocessing	`file.c` → `file.i`
Compilation	`file.i` → `file.s`
Assembly	`file.s` → `file.o`
Linking	`file.o` → `file`

In the case of `gcc(1)`, which we will specifically look into thereafter, three separate programs are used:

- `cc1`: the *real* C compiler, also in charge of preprocessing;
- `as`: the assembler;
- `ld` (called by `collect2` internally): the linker.

In order to make life easier for programmers, `gcc` calls all of these different tools by itself, by passing each intermediate file to the following element of the *pipeline*. Options passed to each of these programs can be displayed by adding the `-v` flag (for "verbose") to the compile line.

Going further...

Historically, the preprocessing phase was performed separately by `cpp(1)` (the "C PreProcessor"), but this is not the case any more. It is still possible to ask `gcc` to produce the intermediate files out of each step of the toolchain (including preprocessing) by giving it the `-save-temps` option. These intermediate files can in turn be given back to the input of `gcc` that will then complete the missing steps of the compilation chain (based on their extension) to produce the final binary.

We will now compile a test program and focus on the different stages of the chain to better understand how they work. Here are our three starting files:

`test.h`:

```
#ifndef TEST_H
# define TEST_H

# define TEST_VAL_1 1
# define TEST_VAL_2 2

int test_func_2(void);

#endif /* !TEST_H */
```

`test.c`:

```
#include "test.h"   // needed for TEST_VAL_1 and TEST_VAL_2

static int test_func_1(void)
{
    return TEST_VAL_1;
}

int test_func_2(void)
{
    return test_func_1() + TEST_VAL_2;
}
```

`main.c`:

```
#include <stdio.h>  // needed for printf()
#include "test.h"   // needed for test_func_2()

int main(void)
{
    int ret = test_func_2();

    printf("%d\n", ret);

    return 0;
}
```

The compilation line used to generate the `test` binary as well as the intermediate files could be:

```
42sh$ gcc -save-temps -o test test.c main.c
```

4.3 Preprocessing

4.3.1 Introduction

The job of the preprocessor is mainly to expand the macros and resolve the files to include in a **translation unit**. A **translation unit** is essentially the `file.i` file resulting from the preprocessing phase, ready to be passed to the compiler (`cc1`). This is a valid C file without preprocessor instructions which contains all the code from included files.

To produce this translation unit, the preprocessor will parse the given input source file (`file.c`) looking for *preprocessor instructions*, which are lines starting with a '#'. These instructions will specify what to do to the preprocessor:

Going further...

The C preprocessor syntax is independent from the C syntax. Therefore you can often get away with using cpp (C PreProcessor) on files which are not C source files.

4.3.2 #define

Replace all occurrences of a word by some text.

Example:

```
#define VALUE 2 // Replace VALUE by 2, everywhere in the code
#define TEST_H  // Replace TEST_H by nothing (just declare it exists)
```

Tips

By convention, macros are always uppercase.

The preprocessor predefines some standard macros (`__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, etc). You can list these macros and their values with the following command:

```
42sh$ gcc -dM -E - < /dev/null
```

Going further...

You can also define a macro using the -D option of gcc.

4.3.3 #include

Include the content of another file in place of the instruction.

Example:

```
#include <stdio.h>
#include "test.h"
```

It has two variants:

- `#include <filename.h>`:

  The preprocessor searches for a file named *filename.h* in a standard list of system directories (/usr/include, ...). You can prepend directories to this list with the -I option. This variant is often used for standard library header.

- `#include "filename.h"`:

  The preprocessor searches for a file named *filename.h* first in the directory containing the current file; if not, found the preprocessor searches for *filename.h* the same way it would have with a `<filename.h>` include style.

Be careful!

You must not use paths that go up in the filesystem tree like `../../filename.h` as it is not considered a good pratice.

Going further...

Included files will also be preprocessed recursively before being included in the translation unit.

Sometimes, when reading the man page of a function, you will notice that it requires some macro to be defined, such as `_POSIX_C_SOURCE`, `_DEFAULT_SOURCE` or `_GNU_SOURCE`. These macros are called **feature test macros** (FTM) and are used to enable specific functions in our libc. FTMs should be defined before including the header file in which the function we want to enable is defined. That way, when the preprocessor expands our `#include` directive, it will conditionally define a set of functions corresponding to the FTM we defined. For more information about FTMs you can read `feature_test_macros (7)`.

Beware, unless specified **explicitly** in the subject, you may not use any FTMs to enable functions.

4.3.4 Conditionals

- `#ifdef MACRO`: Take this branch if the `MACRO` is defined.
- `#ifndef MACRO`: Take this branch if the `MACRO` **is not** defined.
- `#if expression`: Take this branch if the `expression` evaluates to true.
- `#else`: Take this branch if the previous conditional failed.
- `#elif expression`: Take this branch if the previous conditional failed and the `expression` evaluates to true.
- `#endif`: End of the conditional.

4.3.5 Include guard

What happens if two header files include each other, for example:

`test_1.h`:

```
#include "test_2.h"

#define RET_VAL_1 1
```

`test_2.h`:

```
#include "test_1.h"

#define RET_VAL_2 (RET_VAL_1 + 1)
```

`main.c`:

```
#include "test_1.h"

int main(void)
{
    return RET_VAL_2;
}
```

which can be compiled with:

```
42sh$ gcc -o test main.c
```

The preprocessing phase fails because of an infinite recursion.

Include guards use some of the previously explained preprocessor instructions to prevent including the same file more than once in a single *translation unit*. To do so, you have to surround the content of the file with a guard:

```
#ifndef TEST_1_H
# define TEST_1_H

# include "test_2.h"

# define RET_VAL_1 1

#endif /* !TEST_1_H */
```

```
#ifndef TEST_2_H
# define TEST_2_H

# include "test_1.h"

# define RET_VAL_2 (RET_VAL_1 + 1)

#endif /* !TEST_2_H */
```

The preprocessing of the `main.c` file is done this way:

1. The `main.c` file contains the `#include "test_1.h"` instruction. The `test_1.h` file is included for the first time, the `TEST_1_H` macro is not yet defined.
2. The preprocessor enters the condition and defines the `TEST_1_H` macro.
3. The `test_1.h` file contains the `#include "test_2.h"` instruction. The `test_2.h` file is included for the first time, the `TEST_2_H` macro is not yet defined.
4. The preprocessor enters the condition and defines the `TEST_2_H` macro.
5. The `test_2.h` file contains the `#include "test_1.h"` instruction. The `test_1.h` file is included for the second time, the `TEST_1_H` macro is already defined and the preprocessor does not enter the condition.

Be careful!

You may have noticed, the lack of guards is not often fatal. Indeed, the C standard allows *redeclarations* of prototypes or macro as long as they remain identical. The real advantage of guards is to prevent *redefinitions* within headers. However, the use of guards is **mandatory** in all your header files (cf. coding style).

Going further...

As you will notice soon enough, there are plenty of ways to mess up an include guard:

- copy pasting it and forgetting to change the guard variable
- not defining the same variable as the tested one

It's so easy to fail compilers added a fancy feature to replace traditional include guards.

Adding `#pragma once` at the beginning of your file is a safer alternative to previously shown include guards. This directive tells the compiler to only include the header file once, just like what include guards are meant to do!

Beware, it is **not allowed** by EPITA's C Coding Style as it is not part of the standard, nor the most common way to write include guards in C.

4.3.6 Back to our example!

Let's see what happened to our files during the preprocessing phase.

There are two `*.c` files passed as arguments to `gcc`, so there are **two distinct translation units**.

```
42sh$ gcc -E -o test.i test.c
42sh$ gcc -E -o main.i main.c
```

Going further...

gcc -E is used to show the result of a C file preprocessing on the standard output.

The first thing you may notice is that there are many blank lines! Indeed, the lines containing the preprocessing directives are not guarded. It's the same for comments. The lines starting with a '#' in the `*.i` file serve to give the compiler (`cc1`) information on the origin of the following lines. For

example, it can be useful when an error happens to display a message containing the name and line number of the original file.

As expected, the `TEST_VAL_1` macro was extended and the `test.h` containing the two function declarations has been included in the translation unit:

```
# 1 "test.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 31 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 32 "<command-line>" 2
# 1 "test.c"
# 1 "test.h" 1




int test_func_2(void);
# 2 "test.c" 2

static int test_func_1(void)
{
    return 1;
}

int test_func_2(void)
{
    return test_func_1() + 2;
}
```

The `main.c` compilation unit is more than 600 lines long! This is due to the inclusion of the `stdio.h` standard header (`/usr/include/stdio.h`) required to get the declaration of the `printf` standard function before calling it. Similarly, the `test.h` header is required for the declaration of `test_func_2`. The simplified `main.i` file (keeping only the two necessary statements) would look like this:

```
extern int printf (const char* __restrict __format, ...);
int test_func_2(void);

int main(void)
{
    int ret = test_func_2();

    printf ("%d\n", ret);

    return 0;
}
```

Tips

You can ignore `printf`'s prototype which may seem a little bit obscure for the moment. We will come back to the meaning of `extern` later.

5 Macro conditionals

Write a function `hello_word` that prints `Hello world.` to standard output unless the macro `FRENCH` is defined. If so, it should print `Bonjour le monde.`.

```
42sh$ gcc -Wall -Wextra -Werror -pedantic -std=c99 main.c -o main
42sh$ ./main
Hello world.
42sh$ gcc -DFRENCH -Wall -Wextra -Werror -pedantic -std=c99 main.c -o main
42sh$ ./main
Bonjour le monde.
```

6 More about the toolchain

6.1 More macros

There is a lot more to know about macros. You will learn more about it later on. For now, keep in mind that it is useful to define constants and to have *include guards*.

6.2 Compilation

This is the main phase of the toolchain (and also the most complicated): `cc1` takes the content of the *translation unit* (the `*.i` coming out of the preprocessor), parses it and generates the corresponding assembly code. The assembly code is unique to a given processor *architecture*, in our case it is the *x86-64* architecture.

Again, we can ask `gcc` to stop its work right after the compilation phase and provide us with the intermediate assembly code. For this, we use the `-S` option:

```
42sh$ gcc -S -o test.s test.i
42sh$ gcc -S -o main.s main.i
```

Since the `test.s` file is not very interesting, let's rather focus on the `main.s` file. One can observe the body of the `main` function and the calls (*call* ... instruction) to `test_func_2` and `printf`.

```
    .file   "main.c"
    .text
    .section    .rodata
.LC0:
    .string "%d\n"
    .text
    .globl  main
    .type   main, @function
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
```

(continues on next page)

(continued from previous page)

```
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    call    test_func_2@PLT
    movl    %eax, -4(%rbp)
    movl    -4(%rbp), %eax
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   main, .-main
    .ident  "GCC: (GNU) 8.2.0"
    .section    .note.GNU-stack,"",@progbits
```

6.3 Assembly

The assembly instructions (`mov`, `add`, `call`, ...), which are for now in a text file (`*.s`) must be translated into machine language the processor can understand (formed by `0` and `1`). We call these files **object files** (`*.o`). Once again, `gcc` provides us with an option to stop right after this step:

```
42sh$ gcc -c -o test.o test.s
42sh$ gcc -c -o main.o main.s
```

Going further...

You have seen that we used the option `-o` to define the output file name. But with options like `-S` or `-c` with `gcc` you can actually omit this, by default the base name will be kept with the new extension.

Thus:

```
42sh$ gcc -S main.c     // Will produce: main.s
42sh$ gcc -c main.c     // Will produce: main.o
```

The option `-o` would only be useful in this case if we wanted a different name.

Naturally, these files are not directly readable by a human, but we can still use the `objdump` command to *disassemble* the code (meaning to transform the binary code back to assembly):

```
42sh$ objdump -d main.o
```

```
main.o:     file format elf64-x86-64
```

(continues on next page)

(continued from previous page)

```
Disassembly of section .text:

 0000000000000000 <main>:
    0:   55                      push   %rbp
    1:   48 89 e5                mov    %rsp,%rbp
    4:   48 83 ec 10             sub    $0x10,%rsp
    8:   e8 00 00 00 00          callq  d <main+0xd>
    d:   89 45 fc                mov    %eax,-0x4(%rbp)
   10:   8b 45 fc                mov    -0x4(%rbp),%eax
   13:   89 c6                   mov    %eax,%esi
   15:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi        # 1c <main+0x1c>
   1c:   b8 00 00 00 00          mov    $0x0,%eax
   21:   e8 00 00 00 00          callq  26 <main+0x26>
   26:   b8 00 00 00 00          mov    $0x0,%eax
   2b:   c9                      leaveq
   2c:   c3                      retq
```

We find almost the same assembly code as from the previous step. It is now the time to link these two object files (`test.o` and `main.o`) together and produce an executable!

6.4 Linking

The role of the linker is to bring together several object files (`*.o`) into a single binary file. The target file can either be an object file, a standard executable or a library.

An object file contains multiple **symbols**. These are *metadata* used to indicate where in the file are set (stored) some parts of the code, for example, functions or global variables. These symbols can be of different types, but we will focus on the functions.

Each symbol also has a *visibility*. In C, a *global symbol* will be added by the compiler for each function that is defined in the translation unit and has an *extern visibility*. External visibility (or external linkage) is the default visibility for functions and can be specified explicitly using the key word `extern` (remember the declaration of printf in `stdio.h`). To limit the visibility of a function in a translation unit, it is necessary to preface its statement (which can also be its definition) by the `static` keyword:

```c
/* This function can only be called from within the same translation unit */
static int hidden(void)
{
    return 0;
}
```

If we look more closely at the assembly `main.s` file given by the compiler, we can notice the presence of a *.globl main* directive (on line 6) which tells the assembler (`as`) that it will have to create a global symbol for the `main` function in the `main.o` file.

However, if you compile the code containing the `hidden` function, no *.globl hidden* directive will be generated and the assembled object file will not contain a global symbol for `hidden`. It will therefore be unusable by other object files.

There is also a utility called `nm` that displays the list of referenced symbols in an object file. The letter preceding the symbol name informs us about its type; if it is a `T`, the symbol is a function with extern

visibility, if it is a `t` the symbol is a function with static visibility. In both cases the object file contains the code for the function. If it is a `U` (for *undefined*), it is not defined in the file:

```
42sh$ nm test.o main.o
```

```
test.o:
0000000000000000 t test_func_1
000000000000000b T test_func_2

main.o:
                 U _GLOBAL_OFFSET_TABLE_
0000000000000000 T main
                 U printf
                 U test_func_2
```

Going further...

You can use the `-u` option to show only undefined symbols.

The linker will scan object files and attempt to link undefined symbols (`my_func` and `printf` in `main.o`) with their definition. Once the linking is done, an executable or shared library is produced.

The following line produces the final executable from the two object files:

```
42sh$ gcc -o test test.o main.o
```

Some symbols, however, are resolved at runtime when the program launches. This is, for example, the case of `printf` which is not present in the final executable despite the fact that our object files are linked automatically to libc (`libc.so`) by `gcc`.

```
42sh$ nm test | grep printf
```

```
U printf@@GLIBC_2.2.5
```

`gcc` will also link our `*.o` files to other objects that will for example provide the actual entry point of the program, which will be responsible for calling the `main` function and pass its return value to `exit(3)`.

```
42sh$ ./test
3
```

7 Libraries

It is common to have multiple `C` files containing many functions sharing a common goal (ex: a set of functions for manipulating a data type). As an example, it would be absurd to have to re-code functions to handle complex numbers every time we want to use them. One might of course copy different files containing the said `C` functions and add them to each project. Fortunately, there is a lighter and faster alternative.

A library, in programming, is very similar to a “normal” library: it contains a collection of functions (of various sizes). It allows centralizing functions, and grouping them in a homogeneous and coherent

way. Moreover, the distribution of these functions to the public is simplified, as well as its integration within various other projects.

7.1 Static libraries

A static library is nothing more than an archive containing *objects files* (**.o*).

7.1.1 Creation

We must first create the *object files* corresponding to the sources that are to be grouped into a library, obtained after the *Assembly* phase.

Remember, each source file will go through the following phases of the toolchain:

Preprocessing	`file.c` ⇒ `file.i`
Compilation	`file.i` ⇒ `file.s`
Assembly	`file.s` ⇒ `file.o`
Linking	`file.o` ⇒ `file`

Then simply use the `ar` program (the ancestor of `tar`) with some options to create an `ar`chive containing our object files:

```
42sh$ ls
filea.o  fileb.o  filec.o
42sh$ ar csr libdemo.a filea.o fileb.o filec.o
42sh$ ls
filea.o  fileb.o  filec.o  libdemo.a
42sh$ file libdemo.a
libdemo.a: current ar archive
```

`libdemo.a` is a static library!

Going further...

Type *man ar* for more informations!

- The '`c`' option is the silent mode.
- The '`s`' option creates an index into the archive.
- The '`r`' option adds (or replaces) the object files into the archive.

7.1.2 Usage

To use the newly created library in another project, you must specify to the linker the libraries to use (`-l`) and the folders containing them (`-L`) if they are not in the standard system folders. Note that the `-l` option alone adds the *'lib'* prefix to the file name.

Be careful!

You must give the path to the library or its name using the `-l` option **after** the object files that use it.

```
42sh$ ls
main.o  libdemo.a
42sh$ gcc -o bin main.o -L. -ldemo
42sh$ ls
main.o  libdemo.a  bin
```

Note that the previous `gcc` command would have produced the same `bin` executable if the `libdemo.a` file had been explicitly provided to the linker, the `-l` notation is a simple shortcut.

7.2 Shared libraries

7.2.1 Limitations of static libraries

We have seen that to share modules (coherent sets of functions), you could create static libraries. The main problem of these is that they are increasing the size of the binaries, as well as making it harder to produce patches to applications using them. Indeed, to create a binary based on a *static* library, the linker will copy the entire object files stored in the static library into the final binary file (or at least the code of object files whose symbols are referenced elsewhere). By doing so, after every change to a library, you have to invoke the linker on *all* binaries using it to update them. If a library is shared by many programs (e.g. `libc`), this obligation is becoming a major handicap.

To solve this problem, *shared* libraries are used to "delay" part of the linking. It will finalize the linking when launching the executable. During the loading of the executable, the system checks whether the required library is already loaded in memory (already loaded by another program that uses it), and loads the library if it is not already present. Then it finalizes the linking and the binary can run normally. If a new version of the library appears, it will be automatically used by the applications using it after their next launch. There is therefore no need to update binaries.

7.2.2 Creation

By convention, dynamic libraries have the `.so` extension on UNIX (`.dll` on Windows). The **dynamic linker** (`ld-linux.so(8)`), which is invoked just before the start of your code execution, will attempt to find the necessary library in a number of default *paths* (`/usr/lib`, etc.). If the linker cannot find one of the library, then it will fail, preventing the execution of the program.

To create a dynamic library, you have to specify the compiler two flags: - fPIC: making the code relocatable. - shared: making a shared object, linkable.

```
42sh$ ls
file.c
42sh$ gcc -fPIC -shared file.c -o libtest.so
42sh$ ls
file.c libtest.so
```

`libtest.so` is a dynamic library!

7.2.3 Usage

Finally, we can easily link the library to one of our program using its functionalities.

```
42sh$ ls
file.c libtest.so main.c
42sh$ gcc main.c libtest.so -o myprogram
42sh$ ls
file.c libtest.so main.c myprogram
```

Thus, before being able to execute our program, we have to specify where the library can be found with the environment variable LD_LIBRARY_PATH.

```
42sh$ LD_LIBRARY_PATH=/my/path/to/library/folder ./myprogram
Hello world!
```

The only way out is through