Piscine — Tutorial D6

version **#b4173ffc5996c36bf7f9730185ebc8357500bd86**

Assistants C/Unix 2024 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

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

1 Toolchain

1.1 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 limited. 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 0s and 1s 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.

1.2 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 anymore. 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:

File: `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 */
```

File: `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;
}
```

File: `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
```

Tips

-save-temps generates all the intermedia

1.3 Preprocessing

1.3.1 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.

1.3.2 #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 will learn more about it later in this tutorial). 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.

1.3.3 #include

Replace the instruction by the content of another file.

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 do 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 practice. Instead, you can use the -I option as explained above.

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.

1.3.4 Conditionals

In the first tutorial, you briefly learned about **headers guards**. They are actually just a common use of classic preprocessor conditionals that you will discover now:

- `#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 condition failed.

- `#elif expression`: Take this branch if the previous condition failed and the `expression` evaluates to true.
- `#endif`: End of the conditional.

1.3.5 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"
```

(continues on next page)

(continued from previous page)

```
# 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).

Be careful!

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 that compilers added a fancy feature to replace traditional include guards.

Going further...

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 part of the *C* standard, nor the most common way to write include guards. Furthermore, it is forbidden by the EPITA C Coding Style.

1.3.6 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:

```c
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.

1.3.7 Exercise: Macro Conditionals

Goal

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

2 More about the toolchain

2.1 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
    .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
```

As you can see, the assembly corresponding to the functions *test_func_1* and *test_func_2* is not in the same translation unit than the *main* function. It will be resolved later by the linker.

2.2 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



 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
```

This probably does not really mean anything to you, which is okay. You will learn assembly later in this semester.

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

2.3 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, as you will see later.

An object file contains multiple *symbols*. These are *metadata* used to indicate where some parts of the code are set (stored) in the file, 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 keyword `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.

Going further...

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
```

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
```

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
    ...
```

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 -Wvla -c -o hello_world.o hello_world.c
```

Where the wanted `.o` depends on the corresponding `.c`[1].

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

Considering this, our Makefile now will look like this:

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

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.

[1] If you want to produce a `jell.o` from a `plou.c` you are a strange person.

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 -Wvla -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.

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. This is really basic and you should seek to improve upon it as you learn to master `make`.

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

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`[2], ...

Be careful!

When using `make clean` as it can be really easy to accidentally erase all your source code if your `clean` rule is not setup properly. A good way to make sure of what you are doing is to do a dry run (which will only display commands and run them) using `make clean -n`.

```
42sh$ make
gcc -std=c99 -pedantic -Wextra -Wall -Werror -Wvla   -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
```

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

3.3 Setting variables

There are different ways to set variables in a Makefile.

3.3.1 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?
```

3.3.2 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 favorite 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.

3.3.3 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 =.

3.4 Value assignment

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

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

3.4.1 During invocation

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

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

3.4.2 In the environment

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

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

3.5 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

```
main: vector.o list.o
    gcc -o $@ $^
```

3.6 .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 -Wvla

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 -Wvla

.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[3].

4 Libraries

It is common to have multiple *C* files containing many functions sharing a common goal (for instance, 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 real "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.

4.1 Static libraries

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

4.1.1 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!

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

Going further...

Type *man ar* for more information!

- 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.

4.1.2 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.

4.2 Shared libraries

4.2.1 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.

4.2.2 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 two flags to the compiler: - `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!

4.2.3 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!
```

5 Macro Advanced

5.1 The `stringify` operator

The unary operator `#` is called the `stringify` operator because it converts a *macro argument* into a string. If a parameter appears with a prefixed `#`, the preprocessor places the argument between `"`, with slight changes:

- Any sequence of whitespace characters between tokens in the argument value is replaced by a single space character.
- A `\` is prefixed to each `"` in the argument.
- A `\` is prefixed to each `\` that occurs in the argument, except if it introduces a universal character (ex: `\u03B1`).

```c
#define PRINT_EXP(Exp) printf(#Exp " = %lf\n", Exp)

PRINT_EXP(     4.0     *    10.0      );
```

The previous block is expanded as:

```c
printf("4.0 * 10.0" " = %lf\n", 4.0 * 10.0);
```

5.2 The token-pasting operator

The binary operator ## joins its left and its right operands together into a single token. Whitespace characters that appear before and after ## are removed along with the operator itself.

```c
#define JOIN_INT(A, B) A ## B

JOIN_INT(123   ,   456)
```

The previous block is expanded as:

```c
123456
```

Be aware that if you want to append 2 strings, the ## cannot be used directly. And if you need to mix # and ##, you cannot do:

```c
#define JOIN_STR(A, B) #A ## #B
```

This creates 2 strings and then tries to append the strings, thus it fails. You need to do it like this:

```c
#define STR(A) #A
#define JOIN_STR(A, B) STR(A ## B)
```

5.2.1 Exercise: Macro declare and set

Goal

Create a macro `DECLARE_AND_SET(TYPE, NAME, VALUE)`, that declares and sets 3 variables.

- the first one must be a variable with the given type, name and value.
- the second one must be a pointer on the first variable, with the same name as the first one, preceded by *ptr_*.
- the third one must be a string of the given value, with the same name as the first variable, preceded by *str_*.

`DECLARE_AND_SET(int, foo, 42);` should create `int foo = 42;`, `int *ptr_foo = &foo;` and `char *str_foo = "42";`

5.3 Macros using macros

After argument substitution and execution of the # and ## operations, the preprocessor examines the resulting replacement text and expands any macros it contains. But macros cannot be expanded recursively.

```
#define TEXT_1 "Hello"
#define MSG(A) puts(TEXT_ ## A)

MSG(1);
```

The previous block is expanded as:

```
puts("Hello");
```

5.4 Scope and redefinition

You cannot declare two macros with the same name, unless the new replacement text is identical to the existing macro definition. If the macro has parameters, the new parameter names must also be identical to the old ones.

The following sample is valid.

```
#define BAR(X) X
#define FOO(X) X
#define BAR(X) X
```

But not this one:

```
#define BAR(X) X
#define FOO(X) X
#define BAR(x) x
```

However, it can happen that you need to change the meaning of a macro (but it is not a good idea). To change the meaning of a macro, you first need to cancel its current definition using the following directive:

```
#undef MACRO_NAME
```

After that, the identifier `MACRO_NAME` is available for a new macro definition. If `MACRO_NAME` is not a name of a macro, the preprocessor ignores the directive.

The scope of a macro ends with the first `#undef` directive with its name, or if no `#undef` are found, it ends at the end of the translation unit in which it is defined.

Going further...

There is a special situation in which *#undef* is considered to be the way: X-macros. It is a fairly advanced usage of macro that you don't need to understand now, but we strongly encourage you to come back to it later.

5.5 Predefined macros

Compilers that conform to the *ISO C* standard must define the following macros (and some others). Each of these macro names begins and ends with two underscore characters. We talked about them briefly earlier, it is time to see what they actually do:

- __DATE__

  The replacement text is a string literal containing the compilation date in the format "Mmm dd yyyy" (example: Sep 13 2017). If the day of the month is a single-digit number, an additional space character fills the empty space.

- __FILE__

  A string literal containing the name of the current source file.

- __LINE__

  An integer constant whose value is the number of the line in the current source file that contains the __LINE__ macro reference, counting from the beginning of the file.

- __TIME__

  A string literal that contains the time of compilation, in the format "hh:mm:ss" (example: "00:04:02").

Tips

Some macros are predefined only under certain conditions (example: __STDC_NO_COMPLEX__ defined to 1 if the implementation does not support complex numbers, that is if the header complex.h is absent).

Going further...

You can also define macros during compilation with the `-D` flag:

```
gcc -Dwhile=if main.c -o main
```

It can actually be useful with other preprocessor directives like `#ifdef`

```
// some code
#ifdef DEBUG
  printf("I'm debuging!\n");
#endif
// some code
```

```
42sh$ gcc -DDEBUG main.c -o main
42sh$ ./main
   ...
I'm debuging!
   ...
```

6 Exercise: Macro Vector

6.1 Goal

You must implement a generic vector using macros.

Vector is a data structure that contains an array, a capacity that corresponds to the maximum of elements that you can insert in the array, and finally the size corresponding to the number of element already inserted. When you want to insert a new element in a vector that reach its maximum capacity, you need to increase the array size, for example by doubling the array's capacity.

The macros will declare the vector structure as well as its functions.

6.2 Preamble

- Manage your memory wisely, the exercise will be graded checking memory leaks.
- Do not forget to mark your functions as `static inline` or you may end up with duplicated symbols.
- The vector must be able to store any type. Check that pointers, structures and unions work well.
- If memory allocation fails, and the case is not specified, `abort(3)` must be called.
- You can use `gcc -E` to display the output of the preprocessor.
- All macros must take two parameters:
  - `Name`: The name of the structure
  - `Type`: The type stored inside the vector

6.3 Implementation

You must write a file named `vector.h` containing all macros described below.

The vector's initial capacity and resize policy is up to you.

Tips

- You **MUST** declare all functions described below. If you do not, your code will not compile.
- Pay attention to `DECL_VECTOR_GET`, our tests will rely on this macro to read your vector's content.
- Be careful with initialization and free functions.

It will be used like so:

```
DECL_VECTOR_STRUCT(my_int_vector, int)
DECL_VECTOR_INIT(my_int_vector, int)
DECL_VECTOR_INSERT(my_int_vector, int)

struct my_int_vector vector;
```

(continues on next page)

(continued from previous page)

```
my_int_vector_init(&vector); // vector is empty
my_int_vector_insert(&vector, 0, 13); // vector contains: [13]
my_int_vector_insert(&vector, 1, 42); // vector contains: [13, 42]
my_int_vector_insert(&vector, 0, 21); // vector contains: [21, 13, 42]
```

6.3.1 DECL_VECTOR_STRUCT

Declare a macro named `DECL_VECTOR_STRUCT` that declares the vector structure containing:

- Data array
- Vector's size
- Vector's capacity

The macro:

```
DECL_VECTOR_STRUCT(Name, Type)
```

Example usage:

```
DECL_VECTOR_STRUCT(my_int_vector, int)
```

6.3.2 DECL_VECTOR_INIT

Declare a macro named `DECL_VECTOR_INIT` that declares a function that initializes the fields of the previously declared structure.

```
DECL_VECTOR_INIT(Name, Type)
```

For example this macro expansion:

```
DECL_VECTOR_INIT(my_int_vector, int)
```

Must declare and define the following function:

```
static inline void my_int_vector_init(struct my_int_vector *vector);
```

Tips

The function name is the concatenation of `Name` given as macro argument, and _init.

This function must initialize all the structure's fields.

Be careful!

You **MUST NOT** call `free` on `vector`. You do not have the guarantee that the pointer has been allocated using `malloc`. Initialization does not mean neither declaration nor allocation.

6.3.3 DECL_VECTOR_FREE

Declare a macro named `DECL_VECTOR_FREE` that declares a function that releases memory held by the vector but **NOT** the vector itself.

```
DECL_VECTOR_FREE(Name, Type)
```

For example this macro expansion:

```
DECL_VECTOR_FREE(my_int_vector, int)
```

Must declare and define the following function:

```
static inline void my_int_vector_free(struct my_int_vector *vector);
```

6.3.4 DECL_VECTOR_SIZE

Declare a macro named `DECL_VECTOR_SIZE` that declares a function that returns the number of elements held by the vector.

```
DECL_VECTOR_SIZE(Name, Type)
```

For example this macro expansion:

```
DECL_VECTOR_SIZE(my_int_vector, int)
```

Must declare and define the following function:

```
static inline size_t my_int_vector_size(const struct my_int_vector *vector);
```

Note that `vector` must be const qualified.

6.3.5 DECL_VECTOR_INSERT

Declare a macro named `DECL_VECTOR_INSERT` that declares a function that will insert an element at a given position in the vector.

```
DECL_VECTOR_INSERT(Name, Type)
```

For example this macro expansion:

```
DECL_VECTOR_INSERT(my_int_vector, int)
```

Must declare and define the following function:

```
static inline bool my_int_vector_insert(struct my_int_vector *vector, size_t pos, int␣
↪element);
```

When an element is inserted in `n-th` position, all elements starting at the `n-th` position until the last one must be moved to make space for the new element to be inserted.

Example:

```
DECL_VECTOR_STRUCT(my_int_vector, int)
DECL_VECTOR_INIT(my_int_vector, int)
DECL_VECTOR_INSERT(my_int_vector, int)

struct my_int_vector vector;

my_int_vector_init(&vector); // vector is empty
my_int_vector_insert(&vector, 0, 13); // vector contains: [13]
my_int_vector_insert(&vector, 1, 42); // vector contains: [13, 42]
my_int_vector_insert(&vector, 0, 21); // vector contains: [21, 13, 42]
```

The case when the position is strictly greater than the vector's size will not be tested.

This function must return false when memory allocation fails, true otherwise. Note that you must include `stdbool.h` in order to have booleans.

6.3.6 DECL_VECTOR_PUSH_BACK

Declare a macro named `DECL_VECTOR_PUSH_BACK` that declares a function that will insert an element at the end of the vector.

```
DECL_VECTOR_PUSH_BACK(Name, Type)
```

For example this macro expansion:

```
DECL_VECTOR_PUSH_BACK(my_int_vector, int)
```

Must declare and define the following function:

```
static inline bool my_int_vector_push_back(struct my_int_vector *vector, int element);
```

This function must return false when memory allocation fails, true otherwise.

6.3.7 DECL_VECTOR_REMOVE

Declare a macro named `DECL_VECTOR_REMOVE` that declares a function that removes an element from the vector.

```
DECL_VECTOR_REMOVE(Name, Type)
```

For example this macro expansion:

```
DECL_VECTOR_REMOVE(my_int_vector, int)
```

Must declare and define the following function:

```
static inline bool my_int_vector_remove(struct my_int_vector *vector, size_t position);
```

The case when the position is greater or equal to the vector's size will not be tested.

This function must return false when memory allocation fails, true otherwise.

6.3.8 DECL_VECTOR_GET

Declare a macro named `DECL_VECTOR_GET` that declares a function that returns the requested element from the vector.

```
DECL_VECTOR_GET(Name, Type)
```

For example this macro expansion:

```
DECL_VECTOR_GET(my_int_vector, int)
```

Must declare and define the following function:

```
static inline int my_int_vector_get(struct my_int_vector *vector, size_t position);
```

The case when the position is greater or equal to the vector's size will not be tested.

6.3.9 DECL_VECTOR

Declare a macro named `DECL_VECTOR` that declares all the previously described functions.

I must not fear. Fear is the mind-killer.