Exemplos de makefile

```
#
# A simple makefile for managing build of project composed of C source files.
#


# It is likely that default C compiler is already gcc, but explicitly
# set, just to be sure
CC = gcc

# The CFLAGS variable sets compile flags for gcc:
#  -g        compile with debug information
#  -Wall     give verbose compiler warnings
#  -O0       do not optimize generated code
#  -std=c99  use the C99 standard language definition
CFLAGS = -g -Wall -O0 -std=c99

# The LDFLAGS variable sets flags for linker
#  -lm   says to link in libm (the math library)
LDFLAGS = -lm

# In this section, you list the files that are part of the project.
# If you add/change names of source files, here is where you
# edit the Makefile.
SOURCES = demo.c vector.c map.c
OBJECTS = $(SOURCES:.c=.o)
TARGET = demo


# The first target defined in the makefile is the one
# used when make is invoked with no argument. Given the definitions
# above, this Makefile file will build the one named TARGET and
# assume that it depends on all the named OBJECTS files.

$(TARGET) : $(OBJECTS)
    $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)

# Phony means not a "real" target, it doesn't build anything
# The phony target "clean" is used to remove all compiled object files.

.PHONY: clean

clean:
    @rm -f $(TARGET) $(OBJECTS) core
```

Macros

- The line `OBJECTS = $(SOURCES:.c=.o)` defines the OBJECTS macro to be the same as the SOURCES macro, except that every instance of '.c' is replaced with '.o' - that is, this assignment is equivalent to `OBJECTS = demo.o vector.o map.o`.
- There are also two built-in macros used by the makefile, `$@` and `$^`; these evaluate to **demo** and **demo.o vector.o map.o**, respectively, but we will need to learn a bit about targets before we find out why.

```
demo : demo.o vector.o map.o
    gcc -g -Wall -o demo demo.o vector.o map.o -lm

.PHONY: clean

clean:
    @rm -f demo demo.o vector.o map.o core
```

Targets

```
target-name : dependencies
    action
```

The target name is generally the name of the file that will be produced when this target is built. The first target listed in a makefile is the default target, meaning that it is the target which is built when make is invoked with no arguments; other targets can be built using make [target-name] at the command line.

```
[filename].o : [filename].c
    $(CC) $(CFLAGS) -o [filename].o [filename].c
```

Phony targets Note that the clean target in our sample Makefile doesn't actually create a file named 'clean', and thus doesn't fit the pattern which we've been describing for targets. Rather, the clean target is used as a shortcut for running a command which clears out the project's build files (the '@' at the beginning of the command tells Make not to print it to the terminal when it is being run). We flag targets like this by listing them as "dependencies" of `.PHONY`, which is a pseudo-target that we'll never actually build. When the Make utility encounters a phony target, it will run the associated command automatically, without performing any dependency checks.

link:

- https://www.coursera.org/lecture/introduction-embedded-systems/7-makefiles-part-1-4d7SV
- https://www.coursera.org/browse?source=deprecated_spark_cdp
- https://www.coursera.org/specializations/academic-english

Internal macros

- Internal macros are predefined in make.

- `make -p` to display a listing of all the macros, suffix rules and targets in effect for the current build.

Special macros

- The macro `@` evaluates to the name of the current target.

```
prog1: $(objs)
    $(CXX) -o $@ $(objs)
```

is equivalent to

```
prog1: $(objs)
    $(CXX) -o prog1 $(objs)
```

Suffix rules

A way to define default rules or implicit rules that make can use to build a program. There are double-suffix and single-suffix.

- Suffix rules are obsolete and are supported for compatibility. Use pattern rules (a rule contains character `%`) if possible.
- Doubles-suffix is defined by the source suffix and the target suffix. E.g.

```
.cpp.o:
    $(CC) $(CFLAGS) -c $<
```

> This rule tells make that .o files are made from .cpp files.
>
> `$<` is a special macro which in this case stands for a ".cpp" file that is used to produce a ".o" file.

- This is equivalent to the pattern rule `%.o: %.cpp`

```
%.o: %.cpp
    $(CC) $(CFLAGS) -c $<
```

Command line macros

- Macros can be defined on the command line. E.g. `make DEBUG_FLAG=-g`

How Does Make Work?

- The make utility compares the modification time of the target file with the modification times of the dependency files. Any dependency file that has a more recent modification time than its target file forces the target file to be recreated.

- By default, the first target file is the one that is built. Other targets are checked only if they are dependencies for the first target.
- Except for the first target, the order of the targets does not matter. The make utility will build them in the order required.

Marked text

Marked text

Marked text

some emphasized markdown text

some emphasized markdown text

Marked text

Marked text

Marked text

Status: **Not yet implemented**

text

text

Test

Test

Test

highlighted text

Yellow text.

Marked text

This is some text!

background

background

hello

world

text

colors: https://www.computerhope.com/htmcolor.htm#color-codes

markdown: https://learn.getgrav.org/content/markdown settings: https://diessi.ca/blog/writing-mode-in-vs-code/

text

text

text