DETROYAT Alexis
VINCENT Côme
BODART Maxime

MOCA
Rapport
Semaine 2

Modularité (rappel semaine 1)

Nous avons modularisé l'application (dont le code était intégralement contenu dans le fichier dico.c à l'origine) en 6 catégories :

- **dico_tools** (fichiers .c et .h) est un module de gestion de la structure de dictionnaire, dans lequel se trouvent les fonctions **`insertDico`** et **`addToDico`** ;

- **word_tools** (fichiers .c et .h) est un module de gestion des différentes structures associées aux mots, dans lequel se trouvent les fonctions **`next_word`**, **`compareWord`** et **`incWord`** ;

- **display** (fichiers .c et .h) est un module d'affichage, où se trouvent les fonctions **`displayWord`**, **`displayNodes`** et **`displayDico`** ;

- **serialization** (fichiers .c et .h) est un module dédié à la sérialisation/désérialisation des données, c'est-à-dire à leur conversion entre une structure d'arbre binaire de recherche et une structure linéaire (table de hachage), où se trouvent les fonctions **`hash`**, **`deserializeDico`** et **`serializeDico`** ;

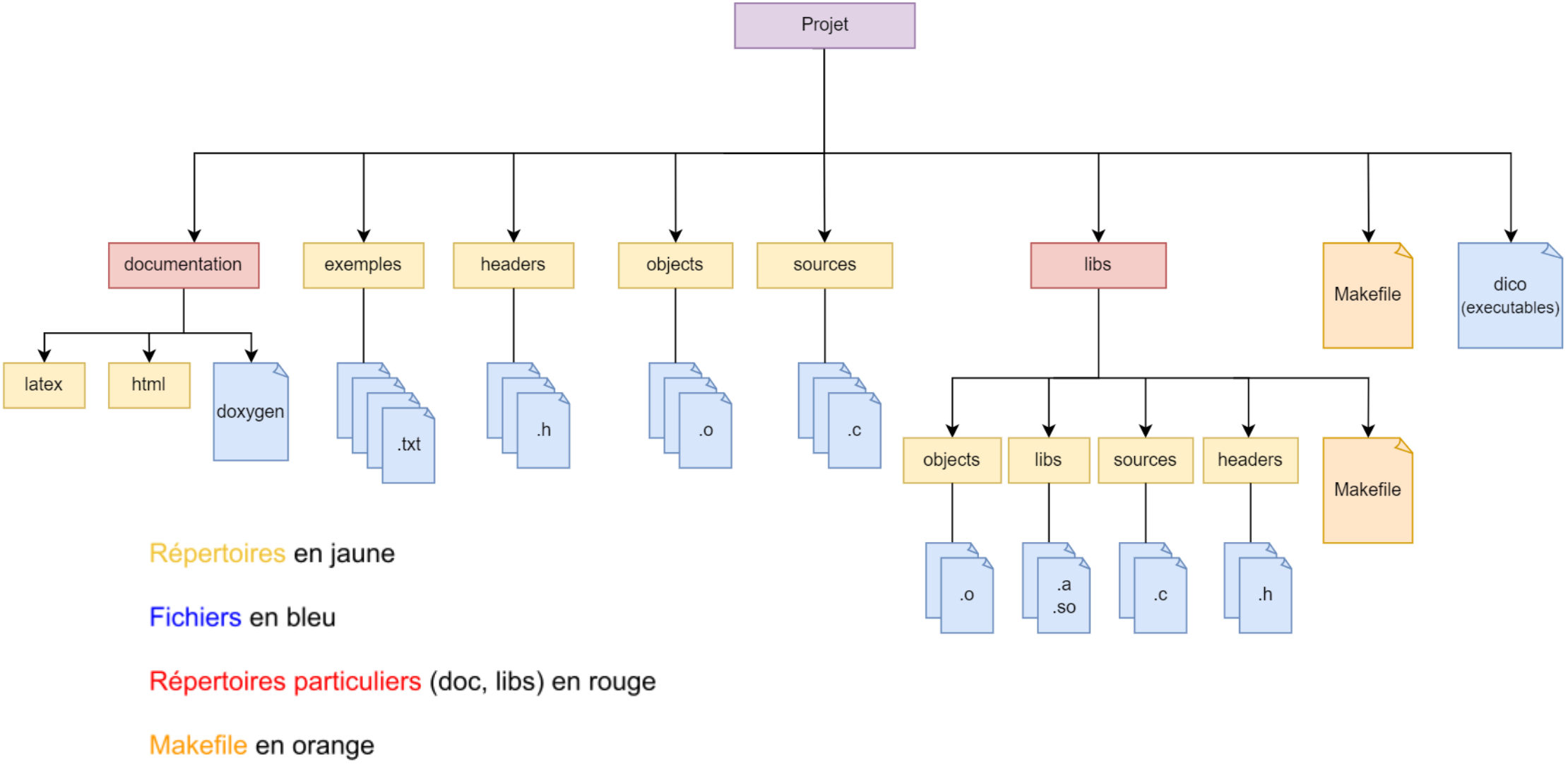
- **structures** (fichier .h uniquement) est un module-header où se trouvent toutes les structures de données utilisées par l'application, soit **`emplacement_t`**, **`mot_data_t`**, **`mot_t`**, **`dico`** ;

- **dico** (fichier .c uniquement) est le module principal de l'application, où se trouve la fonction **`main`** (isolée).

Organisation hiérarchique

Nous avons conservé l'organisation définie en semaine 1 et créé deux librairies, à partir des modules **display** et **serialization** respectivement. En effet, ces deux modules nous semblaient propices à une adaptation sous forme de librairies de par leur réutilisabilité et leur aspect utilitaire (affichage et changement de la structure de données respectivement).

Organisation hiérarchique du projet (semaine 2)



Répertoires en jaune

Fichiers en bleu

Répertoires particuliers (doc, libs) en rouge

Makefile en orange

Nous avons réalisé une hiérarchisation des fichiers du projet permettant une meilleure organisation. À partir du répertoire initial du projet **(`Projet`)**, nous avons organisé le code en **6 sous-répertoires** et **2 fichiers** :

- **`documentation`** est un répertoire contenant la documentation automatique du projet (pour l'instant uniquement en **latex** et **html**) dans des sous-répertoires respectifs, et le fichier de configuration **(`doxy-convert.conf`** simplifié ici par la dénomination **`doxygen`)** qui permet de générer cette documentation ;

- **`exemples`** est un répertoire possédant différents exemples de tests pour le programme principal **`dico`** (tests fournis avec le squelette du projet) ;

- **`headers`** est un répertoire où l'on trouve les différents headers **(`.h`)** des fichiers sources qui ne sont pas dans une bibliothèque ;

- **`objects`** est un répertoire incluant les différents fichiers objets **(`.o`)** créés après la compilation des fichiers sources qui ne sont pas dans une bibliothèque ;

- **`sources`** est un répertoire qui englobe les différents fichiers sources **(`.c`)** du projet autres que ceux qui figurent dans une bibliothèque ;

- **`libs`** est un répertoire où figurent les différentes bibliothèques, elles-mêmes organisées en plusieurs répertoires et régies par un Makefile :

    - **`objects`**, **`sources`** et **`headers`** sont similaires aux sous-répertoires de **`Projet`** mais sont associés aux fichiers des bibliothèques plutôt qu'à ceux du programme principal ;
    - **`libs`** est un répertoire contenant les fichiers de bibliothèques statiques (`.a`) ou partagées (`.so`) ;
    - **`Makefile`** **(dans `libs`)** est le Makefile dédié aux bibliothèques, responsable de la compilation des fichiers sources et de la création des bibliothèques ;

- **`Makefile`** **(dans `Projet`)** est le Makefile dédié au programme principal, responsable de la compilation des fichiers sources, de l'appel au sous **`Makefile`** de **`libs`**, de la création de la documentation du projet et de l'édition de liens pour obtenir l'exécutable **`dico`** ;

- enfin, **`dico`** est l'exécutable de l'application (programme principal).

A l'origine, nous avions créé un répertoire par librairie au sein du répertoire **`libs`** (appelés **`lib_display`** et **`lib_serialization`**), chacun contenant ses propres sous-répertoires **`sources`, `objects`, `headers`,** et **`lib`** pour le(s) fichier(s) **`.a`** ou **`.so`**. Mais compte tenu de la complexité de cette arborescence pour seulement deux librairies (petites, qui plus est), nous avons convenu de nous contenter de répertoires **`sources`, `objects`, `headers`** et **`libs`** communs à toutes les librairies.

Makefile

Comme indiqué ci-dessus, nous avons choisi de séparer le processus de compilation/édition de liens en deux Makefiles distincts : l'un dans le répertoire **`libs`**, dédié aux bibliothèques, et l'autre dans le répertoire **`Projet`**, dédié au programme principal.

La principale motivation derrière ce choix est la structure du projet . Nous souhaitions isoler le processus de compilation des librairies du Makefile principal, afin de pouvoir générer l'exécutable sans recompiler les bibliothèques à chaque **`make`**. Cela nous sera utile lorsque les bibliothèques seront parfaitement fonctionnelles et qu'il sera donc inutile de les regénérer sans cesse. Plus généralement, dans un contexte professionnel, cela permet de créer séparément les librairies, pour ensuite distribuer uniquement les fichiers **`.a/.so`** au client sans les fichiers sources des bibliothèques ni le sous Makefile, tout en garantissant le bon déroulement de la compilation du programme principal. Il suffira alors de supprimer la règle **`lib`** du Makefile principal.

➢ **Makefile des bibliothèques** (`Projet/libs/Makefile`)

```
##################
### VARIABLES ###
##################

CC := gcc
CFLAGS := -Wall -Werror
# If STATIC is undefined, we use the -fPIC (Position Independent Code)
compiler option to be able to generate shared libraries afterwards
# and we add the -lm linker option to get libmath (since we're only going to
use gcc in the case of shared libraries, we won't need -lm for
# static ones)
ifndef STATIC
CFLAGS += -fPIC
LDFLAGS := -lm
endif
# -I option used here to tell the compiler in which folders to search for
the headers (main headers + libs headers)
CPPFLAGS := -I headers -I ../headers

# Directories for source files, object files, and libraries
SRCDIR := sources
OBJDIR := objects
LIBDIR := libs

# Source files for every library
SRCS_DISPLAY := $(SRCDIR)/display.c
SRCS_SERIALIZATION := $(SRCDIR)/serialization.c
# Object files for every library
OBJS_DISPLAY := $(SRCS_DISPLAY:$(SRCDIR)/%.c=$(OBJDIR)/%.o)
```

```
OBJS_SERIALIZATION := $(SRCS_SERIALIZATION:$(SRCDIR)/%.c=$(OBJDIR)/%.o)

# If STATIC is defined, we want to generate static libraries, thus we use
the "ar" command and the ".a" suffix name
# Otherwise, we want to generate shared libraries, thus we use the compiler
(gcc) with appropriate options (-shared -o) and the ".so" suffix name
ifdef STATIC
LIB_EXT := .a
AR := ar rs
else
LIB_EXT := .so
AR := $(CC) -shared -o
endif

# Debug mode : -g to be able to run gdb, and -DDEBUG to define the DEBUG
macro in source files
ifdef DEBUG
CFLAGS += -g
CPPFLAGS += -DDEBUG
endif

# Library names
DISPLAY_LIB_NAME := $(LIBDIR)/libdisplay$(LIB_EXT)
SERIALIZATION_LIB_NAME := $(LIBDIR)/libserialization$(LIB_EXT)

##############
### RULES ###
##############

.PHONY: clean

# Default target
all: ensureDirs $(DISPLAY_LIB_NAME) $(SERIALIZATION_LIB_NAME)

# Rule to make the display library
$(DISPLAY_LIB_NAME): $(OBJS_DISPLAY)
    $(AR) $@ $^ $(LDFLAGS)

# Rule to make the serialization library
$(SERIALIZATION_LIB_NAME): $(OBJS_SERIALIZATION)
    $(AR) $@ $^ $(LDFLAGS)

# Rule to create all needed directories (-p to disable errors if they
already exist)
ensureDirs:
    @mkdir -p $(OBJDIR)
    @mkdir -p $(LIBDIR)
```

```
# Rule to make required object files for the libraries
$(OBJDIR)/%.o: $(SRCDIR)/%.c
    $(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<

# Rule to clean the workspace
clean:
    -rm -r $(OBJDIR) $(LIBDIR)
```

Nous commençons par définir les variables usuelles (`CC`, `CFLAGS`, `CPPFLAGS`, `LDFLAGS`) en ajoutant (via **`ifdef ... endif`**) aux flags de compilation l'option **`-fPIC`** (pour compiler en Position Independent Code, ce qui permet la génération de librairies partagées) et aux flags du linker l'option **`-lm`**, si et seulement si la macro **`STATIC`** n'est pas définie (en ligne de commande donc). Cela correspond au cas où l'utilisateur souhaite générer des librairies partagées (fichiers **`.so`**). Nous ajoutons également aux flags du préprocesseur les options **`-I [dirname]`** permettant d'indiquer où chercher les headers des différents fichiers sources.

Par la suite, nous définissons des variables associées aux noms des différents répertoires liés aux bibliothèques (**`sources`**, **`objects`**, **`libs`**) ainsi que d'autres variables associées aux listes des fichiers sources et objets de chaque bibliothèque.

Vient alors l'étape de choix de la commande à exécuter pour générer les librairies, qui dépend aussi de la macro **`STATIC`** :

- si cette macro est définie, le suffixe du nom de fichier sera **`.a`** et on utilisera la commande **`ar rs`** (option **`s`** permettant d'ajouter un index à l'archive, équivalente à **`ranlib`**) pour créer la bibliothèque en mode **statique** ;

- sinon, le suffixe du nom de fichier sera **`.so`** et on utilisera la commande **`gcc -shared -o`** pour créer la bibliothèque en mode **dynamique**.

Nous dotons le Makefile d'un mode debug (macro **`DEBUG`**) comme en semaine 1, puis nous définissons les variables associées aux noms des fichiers librairies à produire. Parmi les règles écrites :

- **`ensureDirs`** permet de créer les répertoires des fichiers objets et des fichiers librairies s'ils n'existent pas, en utilisant l'option **`-p`** pour désactiver l'erreur provoquée dans le cas où ils existent déjà ;

- **`$(DISPLAY_LIB_NAME)`** et **`$(SERIALIZATION_LIB_NAME)`** sont les règles de création des librairies (via la commande retenue dans l'étape précédente) ; cela implique bien entendu la compilation des fichiers sources avec la règle **`%.o : %.c`**.

➢ **Makefile du programme principal** (Projet/Makefile)

```
#################
### VARIABLES ###
#################

CC=gcc
CFLAGS=-Wall -Werror
# -I option used here to tell the compiler in which folders to search
for the headers (main headers + libs headers)
CPPFLAGS=-I headers -I libs/headers
MAKEARGS=

# -Wl,-rpath,libs/libs/ add a directory to find our libraries (allows us
to omit the use of "export
LD_LIBRARY_PATH=./libs/libs/:$LD_LIBRARY_PATH")
LDFLAGS=-Wl,-rpath,libs/libs/
# If STATIC is defined, we want to generate static libraries, whose name
end with ".a" and force a static link with libs display and
serialization
# Otherwise, we want to generate shared libraries, whose name end with
".so" and force a dynamic link with libs display and serialization
ifdef STATIC
LIBEXT=.a
MAKEARGS+=STATIC=1
else
LIBEXT=.so
endif
# Libraries specific options :
# -L to give the libraries path name
# -l to give the name of libraries files (with the ":" syntax allowing
us to specify the full name, including the right suffix depending on
# whether we want to generate static or dynamic libraries)
LDFLAGS+=-L libs/libs -l:libdisplay$(LIBEXT)
-l:libserialization$(LIBEXT) -lm

# Directories for source files, object files, headers, executable files,
and Doxygen documentation
SRCDIR=sources
OBJDIR=objects
HEADERSDIR=headers
EXECDIR=.
DOCDIR=documentation

# Files names : sources, headers, objects, objects that correspond to an
executable file name, and executable files
SRCS=$(wildcard $(SRCDIR)/*.c)
```

```
HEADERS=$(wildcard $(HEADERSDIR)/*.h)
OBJS=$(SRCS:$(SRCDIR)/%.c=$(OBJDIR)/%.o)
MAIN_OBJS=$(OBJDIR)/dico.o
EXEC=$(MAIN_OBJS:$(OBJDIR)/%.o=$(EXECDIR)/%)
# Debug mode :# -g to be able to run gdb
# -DDEBUG to define the DEBUG macro in source files
# DEBUG=1 option passed to the sub Makefile (used to generate libraries)
to activate debug mode in libraries source files too

ifdef DEBUG
CFLAGS+=-g
CPPFLAGS+=-DDEBUG
MAKEARGS+=DEBUG=1
endif


##############
### RULES ###
##############

.PHONY : clean

# Default target
all : lib ensureDirs message $(EXEC) doc

# Rule to organize the output
message :
    @echo
    @echo "----- Compilation et génération du programme principal -----"

# Rule to make the object files
$(OBJDIR)/%.o:$(SRCDIR)/%.c
    $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $@

# Specification of the headers that every object file depends on (in
case only preprocessor instructions are modified)
$(OBJDIR)/dico_tools.o : $(HEADERSDIR)/dico_tools.h
$(HEADERSDIR)/structures.h $(HEADERSDIR)/word_tools.h
libs/headers/serialization.h
$(OBJDIR)/word_tools.o : $(HEADERSDIR)/word_tools.h
$(HEADERSDIR)/structures.h

# Special case for main objects files because they depend on all headers
$(MAIN_OBJS): $(HEADERS)

# Rule to generate executable files
```

```
$(EXEC) : $(OBJS)
    $(CC) $(OBJS) -o $@ $(LDFLAGS)

# Rule to compile and generate libraries (using the sub Makefile in
libs)
# && is used to execute several commands in a row and in the same shell
(otherwise every process is run on a different shell)
# $(MAKE) is a standard variable containing the make command (used to
avoid conflicts with other potentially defined make variables)
lib:
    @echo "----- Nettoyage et compilation des librairies -----"
    cd libs && $(MAKE) clean && $(MAKE) $(MAKEARGS)

# Rule to generate automatic documentation (via Doxygen, based on the
configuration file located in DOCDIR)
doc:
    @echo
    @echo "----- Création de la documentation dans le repertoire
documentation -----"
    doxygen $(DOCDIR)/doxy-convert.conf

# Rule to create all needed directories (-p to disable errors if they
already exist)
ensureDirs:
    @mkdir -p $(OBJDIR)
    @mkdir -p $(EXECDIR)
    @mkdir -p $(DOCDIR)

# Rule to clean the workspace
# The LIBCLEAN option can be specified in command line to also clean the
libraries (libs) subdirectory (through the sub Makefile)
clean :
    @echo "----- Nettoyage des objets, des exécutables et de la
documentation -----"
    -rm -r $(OBJDIR)/*.o $(EXEC) $(DOCDIR)/*/
ifdef LIBCLEAN
    @echo
    @echo "----- Nettoyage des librairies -----"
    cd libs && $(MAKE) clean
endif
```

Le Makefile du répertoire principal a la charge de la compilation et de l'édition de liens pour générer l'exécutable, en appelant au passage le sous Makefile dédié aux bibliothèques pour compiler celles-ci. Nous commençons par définir les variables usuelles associées aux différents flags (`CC`, `CFLAGS`, `CPPFLAGS`, `LDFLAGS`) en ajoutant dans `LDFLAGS` l'option **`-Wl,-rpath,libs/libs/`** afin de permettre au linker de chercher les fichiers

librairies dans le répertoire spécifié en troisième partie de la commande. Ceci a l'avantage de permettre à l'utilisateur d'exécuter le programme principal sans devoir préalablement définir le chemin de recherche des librairies dans sa variable d'environnement **LD_LIBRARY_PATH**. Comme pour le Makefile précédent, les options **-I** des flags du préprocesseur permettent de spécifier les chemins de recherche des headers.

De la même manière, la macro **STATIC** définit la suite du comportement du Makefile : si elle est définie, on cherchera à inclure des fichiers de bibliothèques suffixés par **.a** et on transmet l'information au sous Makefile via l'argument **STATIC=1**, sinon on cherchera à inclure des fichiers de bibliothèques suffixés par **.so**. Nous ajoutons alors aux flags du linker les options **-L** afin de spécifier le chemin de recherche de nos bibliothèques, ainsi que **-l** permettant d'indiquer les noms des fichiers de bibliothèques (la syntaxe **-l:[name]** est ici utilisée pour entrer le nom complet des fichiers librairies, en incluant leurs suffixes). Seule la librairie système **math** est inluse avec la syntaxe standard, puisqu'elle sera toujours linkée de la même manière.

Nous définissons ensuite les variables de répertoires, et de noms de fichiers (sources, headers, objets, objets principaux, exécutables), avant d'intégrer un mode debug via la macro **DEBUG**. Les règles sont les suivantes :

- **lib** prend en charge l'appel au sous Makefile pour nettoyer le répertoire des librairies puis les recompiler, à noter que ce comportement par défaut pourra être modifié lorsque les fichiers sources des bibliothèques seront fonctionnels, comme indiqué précédemment ;

- **ensureDirs** crée les répertoires requis pour la compilation du programme principal s'ils n'existent pas, comme dans le sous Makefile ;

- **message** est une règle d'affichage, qui structure l'output obtenu ;

- **$(EXEC)** est la règle standard permettant de générer l'exécutable, en exploitant nécessairement la règle de compilation **%.o : %.c**

- **doc** est la règle de génération de la documentation automatique (Doxygen) ;

- **clean** permet de nettoyer le répertoire courant en supprimant les fichiers objets, exécutables, ainsi que la documentation ; avec l'argument **LIBCLEAN=1** en ligne de commande, cette règle appelle en outre la règle **clean** du sous Makefile afin de nettoyer également le répertoire des bibliothèques.