# Make

Gilles Maire

2017

# Plan de la formation

# Make (I)

## Rubriques

- Utilisation
- Autres conventions
- Particularismes

# Utilisation

# Présentation

- Un fichier Makefile permet d'automatiser des tâches de compilation ou d'automatiser les différentes étapes de la fabrication d'un fichier
- On peut donc utiliser des Makefile pour fabriquer de la documentation à partir de convertisseurs, fabriquer une base de données ou bien compiler des programmes
- Le projet Makefile fait partie du projet GNU et on trouve la page d'accueil à l'adresse https://www.gnu.org/software/make
- La documentation complète en langue anglaise est disponible sur la même page à l'adresse https://www.gnu.org/software/make/manuel

# Makefile : Usage

- Un Makefile est un fichier de configuration de la commande make, il s'écrit avec un M majuscule
- Une commande **make** permet d'automatiser le processus de génération d'un fichier résultat
  - au départ on utilisait une commande make pour expliciter les étapes de la compilation : compilation, exécution de lien avec les librairies etc. . .
  - on peut étendre l'utilisation d'un Makefile à la génération d'une documentation ou à tout autre processus automatique de génération d'un fichier.
- En général les Makefile s'appellent avec un argument qui est une des étiquettes du fichier Makefile qui permet de faire plus ou moins les choses suivantes :
  - make : lit la première étiquette
  - make clean : nettoyage sauf les fichiers exécutables
  - make all : compilation
  - make distclean : nettoyage complet y compris les binaires
  - make install : installation (en général dans /usr/local/bin)
  - make uninstall : désinstallation
- Mais cela dépend des étiquettes que le développeur a prévu dans son Makefile.

# Etiquettes et cibles

- À l'intérieur d'un fichier Makefile on trouve des lignes de la forme

```
fichier.o : fichier.c
    gcc fichier.c -o fichier.o
```

- fichier.o est une étiquette ou une cible
- fichier.c est une dépendance
- gcc fichier.c -o fichier.o est une commande
- Ces lignes se lisent comme ceci :
  - Le fichier fichier.o dépend de fichier.c c'est à dire que si fichier.c est plus récent que fichier.o, on doit reconstruire fichier.o
  - pour fabriquer le fichier .o la règle est à la ligne suivante.
- On distingue les étiquettes des lignes de commandes car la ligne de commandes commence par un caractère tabulation

# Makefile : syntaxe

- Un Makefile comprend donc es lignes de la forme :

```
cible: dependance
    commandes
    commandes
```

- Par exemple

```
hello: hello.o main.o
    gcc -o hello hello.o main.o
hello.o: hello.c hello.h
    gcc -o hello.o -c hello.c -W -Wall -ansi -pedantic
main.o: main.c hello.h
    gcc -o main.o -c main.c -W -Wall -ansi -pedantic
clean:
    rm -f *.o
distclean:
    rm  -f *.o hello
```

## Premier exemple

- Écrire un programme en c++ comprenant une classe fournie par un fichier h et un fichier cpp contenant une classe, qui affichera un message ainsi qu'un fichier essai.cpp contenant la fonction main()
- Écrire un Makefile qui génère la compilation et qui contient un clean et un distclean

# Autres conventions

# Makefile : autres conventions

- Une ligne de commandes ou de dépendances peut prendre sur plusieurs lignes, la fin de chaque ligne appelant une suivante sera le caractère anti slash
- Mais plusieurs commandes peuvent se suivre, dans ce cas le retour ligne sera utilisé sans omettre une tabulation devant chaque commande.
- Si une cible est donnée en entrée du Makefile, c'est cette cible qui sera exécutée, si aucune cible n'est donnée, c'est la première cible qui sera exécutée
- Une ligne commençant par # est un commentaire
- Une ligne de commandes commençant par @ verra le résultat de la commande masquée à l'exécution.
- Lorsque la commande make est appelée les fichiers suivants sont utilisés par ordre décroissant de priorité :
  - GNUmakefile, makefile et Makefile
  - Votre fichier peut utiliser un autre nom dans ce cas on utilisera

```
make -f autrenom
```

- On peut inclure un autre Makefile via la commande

```
include nomdufichier
```

## Amélioration du premier exercice

- Reprendre l'exercice précédent mais masquer l'affichage de la commande rm à l'utilisateur
- Ajouter une étiquette launch qui lance votre programme en affichant "lancement du programme exercice" et à la ligne suivante

le résultat de l'exécution

# Makefile : introduction aux variables

- Les lignes :

```
hello: hello.o main.o
    gcc -o hello hello.o main.o
```

- peuvent être remplacées par :

```
objects = hello.o main.o
hello: $(objects)
        cc -o edit $(objects)
```

- On peut utiliser un caractère * pour remplacer une liste de fichiers

```
objects = ../obj/*.o
hello: $(objects)
        cc -o edit $(objects)
```

- On aura tendance à remplacer les * par la fonction wildcard plus permissive si la liste est vide

```
objects= $(wildcard *.o)
```

## Makefile avec des variables

- Ajouter une deuxième classe2 identique à la première classe
- Appeler un deuxième affichage depuis le main en utilisant la deuxième classe
- Adapter le main en mettant une variable représentant l'ensemble des fichiers objets

# Comment make fonctionne

- La commande make va lire le fichier Makefile par défaut et se positionner à la première cible pour voir si elle a des dépendances
- Si elle en a, make regarde la date de chacun des fichiers de dépendances et pour ceux qui sont plus récents, make va commander leur génération par la ou les commandes correspondantes.
- Il faut noter que même si la première ligne est celle qui est regardée en premier, un fichier c peut modifier un fichier o qui redéclenchera la première règle.
- En d'autres terme make s'arrêtera de boucler lorsque tous les fichiers générés à partir de fichiers sources sont plus récents.

# Particularismes

# Comment débuguer un Makefile

- Le fichier Makefile est assez sensible voici les erreurs de syntaxe courantes :
  - pas de tabulation devant une commande
  - si plusieurs commandes à la suite, une tabulation devant chaque commande
  - variables initialisée avec : plutôt qu'avec =

```
HTML : *.a
OBJ: $(HTML)
```

- Aucune règle trouvée
  - Cela se produit quand on intervertit les fichiers destinations et sources

```
*.c : *.o
```

- Pour afficher les variables mettre une commande echo devant les commandes le temps du test

```
OBJ : $(HTML)
    echo  ${HTML}
    makehtml $*
```

# Debuguer un makefile (suite)

- L'option -k de la commande make permet de continuer même si une erreur est rencontrée
- L'option -d affiche des résultat de débug du Makefile (équivalent à —debug=a)
- L'option –debug=option où option peut être :
  - a pour all
  - b pour basic
  - v pour verbose
  - i affiche les messages concernant les variables implicites
  - j affiche les messages concernant les sous commandes

## Debuguer un make

- Lancer le make avec l'option -d puis avec l'option - - debug=b

# Usage récursif

- L'utilisation récurcive signifie qu'un Make va appeler un make, ce qui est utile :
  - pour gérer des Makefile en provenance de plusieurs sous projets
  - pour gérer des sous répertoires à l'intérieur d'un même projet
- On peut utiliser l'une des deux syntaxes aux choix :

```
subsystem:
        cd subdir && $(MAKE)
```

ou

```
subsystem:
        $(MAKE) -C subdir
```

- Seule la variable CIRDIR sera affectée, les autres variables resteront invariantes après chaque passage au Make
- Si on souhaite passer des variables d'un Make à l'autre on doit ajouter pour la variable :

```
export VARIABLE1=valeur
export VARIABLE2
```

# Variables conditionnelles

- On peut positionne une variable si celle-ci n'est pas encore positionnée par l'utilisation du symbole ?=

```
VARIABLE ?=  VALEUR
```

- On peut positionner une variable par le retour du shell par le symbole !=

```
hash !=  printf '\043'
```

- On peut ajouter du texte à une variable déjà définie par

```
objects += another.o
```

- La notation := est similaire += mais étend la variable récursivement c'est à dire en étendant toutes les variables:

```
objects := $(variable) more
```

- Pour toutes ces options la directive override permet de réaffecter une variable passée en argument du make

```
override objects = $(variable)
```

# Make (II)

## Rubriques

- Utilisation plus avancée des variables
- Conditions et fonctions

# Utilisation plus avancée des variables

# Règles implicites

- Un Makefile connaît des règles implicites comme entre autres de produire des exécutables à partir d'un fichier.c ; c'est à dire de faire tout seul la commande :

```
gcc -c main.c -o main.o
```

- Cela revient à dire que nous pouvons écrire :

```
objets= objets1.o objet2.o
all: $(objets)
     gcc -o program $(objets)
objet1.o: objet1.c
objet2.o: objet2.c
```

- Si on utilise make avec l'option -r on désactive ces rôles implicites
- La documentation révèle la liste des règles implicites en fonction des langages

## Règles implicites

- Démontrer que make ne peut trouver de règle implicite pour notre exemple précédent
- Écrire un programme C++ avec règle implicite

# Groupement des cibles

- Cet exemple utilise les règles explicites

```
gcc -c main.c -o main.o
```

```
objects = main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o
edit : $(objects)
       cc -o edit $(objects)
$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

# Makefile : autres variables

```
hello: $(objects)
    cc -o hello $(objects)
```

- **$@ :** Le nom de la cible (hello)

```
cc -o $@ $(objects)
```

- **$<** : Le nom de la première dépendance (hello.o)
- **$^** : La liste des dépendances (hello.o main.o)
- **$?** : La liste des dépendances plus récentes que la cible
- **$*** : Le nom du fichier sans suffixe
- **%.c** : n'importe quoi suivi de .c

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

## Règles génériques

- Optimiser le programme aux deux classes via des variables %.c

# Makefile : optimisation

- Nous voulons générer de la documentation automatique c'est à dire convertir un répertoire de fichiers txt en html via la commande asciidoc
- Une premier approche nous donne :

```
fichier1.hmtl: fichier1.txt
    asciidoc -o fichier1.html fichier1.txt
fichier2.hmtl: fichier2.txt
    asciidoc -o fichier2.html fichier2.txt
fichier3.hmtl: fichier3.txt
    asciidoc -o fichier3.html fichier3.txt
```

- on peut généraliser cela à tous les fichiers html

```
all: fichier1.html fichier2.html fichier3.html
%.html: %.txt
    asciidoc -o $@ $<
```

- Cela marche mais il faut donner la liste des fichiers html et cette liste n'existe pas après un make clean .. Comment faire ?

# Makefile : les fonctions

- il existe un certain nombre de fonctions en plus de la fonction wildcard que nous avions vu. Prenons la fonction subst

```
$(subst ee,EE,feet on the street)
```

- L'exemple précédent peut devenir :

```
HTML=$(subst txt,html,$(wildcard *.txt))
all : $(HTML)
%.html : %.txt
    asciidoc -o $@ $<
    @mv $@ ../html
```

# Makefile : mode enrichi

- Etiquettes argumentaires ( en général clean all et install)
- Souvent précédées en première ligne de l'instruction .PHONY, ceci afin de ne pas les confondre avec des noms de fichier

```
.PHONY: all clean
all: hello
clean:
    rm -rf *.o
```

- make all
- make clean

# Conditions et fonctions

# Exemple compilation conditionnelle

- L'exemple ci-dessous montre l'usage de directives **ifeq else** et **endif**

```
libs_for_gcc = -lgnu
normal_libs =
foo: $(objects)
ifeq ($(CC),gcc)
        $(CC) -o foo $(objects) $(libs_for_gcc)
else
        $(CC) -o foo $(objects) $(normal_libs)
endif
```

- Ce qui aurait pu s'écrire également sous la forme

```
libs_for_gcc = -lgnu
normal_libs =
ifeq ($(CC),gcc)
  libs=$(libs_for_gcc)
else
  libs=$(normal_libs)
endif
foo: $(objects)
        $(CC) -o foo $(objects) $(libs)
```

# Liste des directives

- Les principales directives conditionnelles sont :
  - **ifeq (arg1, arg2)** ou ifeq ‘arg1’ ‘arg2’ ou ifeq “arg1” “arg2” ou ifeq “arg1” ‘arg2’ ou ifeq ‘arg1’ “arg2” : si les arguments sont égaux
  - **ifneq (arg1, arg2)** ou ifneq ‘arg1’ ‘arg2’ ou ifneq “arg1” “arg2” ou ifneq “arg1” ‘arg2’ ou ifneq ‘arg1’ “arg2” : s’ils ne sont pas égaux
  - **ifdef variable** : si variable est définie
  - **ifndef variable** : si variable n’est pas définie
- Exemples :

```
ifdef foo
frobozz = yes
else
frobozz = no
endif

bar = true
foo = bar
ifdef $(foo)
frobozz = yes
endif
```

# Fonctions

- **$(subst depuis,vers,texte)** : substitue le texte **depuis** par **vers** dans le **texte**
- **$(patsubst motif,remplacement,texte)** : trouve dans **texte** un mot séparé par des blancs qui correspond à **motif** et le remplace par **remplacement**

```
$(patsubst %.c,%.o,x.c.c bar.c)
```

- **$(strip chaine)** : supprime les caractères blancs avant et après la **chaine**
- **$(findstring chaine,dans)** : cherche **chaine** dans la chaine **dans** et renvoi la chaine ou une valeur vide (peut être utilisé dans un test)
- **$(filter motif...,texte)** : renvoie tous les mots séparés par des blancs dans texte qui correspondent avec les **motifs** (avec %)

```
sources := foo.c bar.c baz.s ugh.h
foo: $(sources)
        cc $(filter %.c %.s,$(sources)) -o foo
```

- **$(filter-out pattern...,text)** : renvoie tous les mots séparés par des blancs dans texte qui ne correspondent pas avec les **motifs** (avec %)
- **$(sort liste)** : trie la liste de mot

# Fonctions (suite)

- **$(word n,texte)** : retoune le n ieme mot du **texte**
- **$(words texte)** : retourne le nombre de mots dans le texte
- **$(firstword liste)** : retourne le premier mot de la **liste**
- **$(lastword liste)** : retourne le dernier mot de la **liste**
- **$(dir noms)** : retourne le répertoire de chaque **nom** sous forme de liste
- **$(notdir noms )** : retourne le nom de fichier de chaque **nom** sous forme de liste
- **$(suffix noms...)** : retourne la listes des extensions de fichiers pour chaque **nom**
- **$(basename noms...)** : retourne la liste des répertoires / **noms** sans les extentions
- **$(addsuffix suffix,noms...)** : ajoute l'extention **suffix** à chaque **nom** de fichier
- **$(addprefix prefix,noms)** : ajoute le nom de répertoire prefix devant chaque nom
- **$(join list1,list2)** : joint deux listes parallèles
- **$(wildcard pattern)** : remplace le pattern (comme en shell avec *) en noms de fichiers
- **$(realpath names...)** : renvoie le nom asbolue sans . ou .. et sans lien symbolique
- **$(abspath names...)** : idem mais ne résoud pas les liens symboliques
- **$(error text...)** : provoque une erreur en affichant le message
- **$(warning text...)** : provique un warning en affichant le message
- **$(shell command)** : execute la commande en shell et retourne sa valeur
- **$(origin variable)** : renvoie undefined, default, environment, etc... pour indiquer

## Creation d'un make pour pandoc

- Fabriquer un Makefile générique pour fabrication de transparents Pandoc /Markdown
- Un support de cours au format markdown se fabrique via la commande :

```
pandoc -st beamer -V theme:CUSTOM  fichier.md -o fichier.pdf
```

- On fera en sorte que le fichier Makefile soit dans un répertoire NomDuCours, que le fichier source soit NomDuCours.md
- Ainsi le nom du cours sera calculé en fonction du répertoire où se trouve le Makefile.

# En amont

## Rubriques

- Génération des Makefile
- Automake

# Génération des Makefile

# Génération des makefile

- Si on peut écrire manuellement des fichiers Makefile de petite taille, on utilise des outils de génération pour des Makefile plus complexes de plusieurs façons :
  - configure via autoconf et automake
  - qmake, cmake, scon, waf
- Dans tous les cas, ces programmes généreront un Makefile que nous devrons ensuite lancer via la commande

```
make
```

# Configure

- **Syntaxe :**

```
./configure
```

- vérifie la présence des bibliothèques nécessaires, l'architecture et la présence des compilateurs
- Un «gros» configure dure quelques minutes une grosse compilation quelques heures,
- configure permet donc de lancer la compilation en limitant les risques d'échecs dûs à un manque de composant

# Options courantes de configure

- Chaque fichier configure comprend ses propres options mais un certain nombre d'entre elles sont toujours présentes :
  - **-h –help :** fournit la liste des options disponibles
  - **-srcdir=DIR :** indique le répertoire où les sources sont présentes
  - **–prefix=DIR :** par défaut le préfixe d'installation est /usr/local mais on peut le forcer à une autre valeur
  - **–exec-prefix=DIR :** par défaut l'exécutable est généré dans PREFIX/bin mais on peut changer cette valeur
  - **–data-dir=DIR :** par défaut c'est /usr/local/share mais peut être changé pour DIRd
  - **–sysconfdir=DIR :** par défaut /usr/local/etc mais peut être changé pour DIR
  - **–build=xx :** précise l'architecture du serveur de compilation
  - **–host=xx :** demande la cross compilation pour le host xx
  - **–target=xx :** précise la machine sur lequel servira le compilateur (on peut produire sur linux (build) un compilateur pour MacOsx (target) qui produira un code pour un ARM (host), ce cas s'appelle la compilation Canadienne).
  - **CFLAGS :** paramètres à ajouter au compilateur

# Génération par qmake

- Dans le cas des fichiers C++ Qt, on utilise un fichier d'extension pro qui sera capable de générer automatiquement le fichier Makefile par la commande

```
qmake fichier.pro
make
```

- Dans ce cas on apprend les options de génération des fichiers pro propres à Qt plutôt que les options de Makefile
- Quelques variables du fichier pro :
  - TARGET :nom de l'application
  - target.path = /usr/bin
  - INSTALLS += target

# Génération par cmake

- CMake est un sytème de gestion alternatif qui est piloté par un fichier `CMakeLists.txt`

```
cmake .
```

- **Exemple :**
- CMakeLists.txt en cascade :

```
cmake_minimum_required (VERSION 2.6)
project (HELLO)
add_subdirectory (Hello)
add_subdirectory (Demo)
include_directories (${HELLO_SOURCE_DIR}/Hello)
link_directories (${HELLO_BINARY_DIR}/Hello)
add_executable (helloDemo demo.cxx demo_b.cxx)
```

# Automake

# Autoconf

- Autoconf est un outil permettant de générer un fichier configure. IL nécessite d'installer le paquet autoconf
- Autoconf est lié à Automake, à gnulib ( qui gère la glibc) et à Libtool ( qui gère les librairies dynamiques)
- La page de manuel d'autoconf est disponible à l'adresse https://www.gnu.org/savannah-checkouts/gnu/autoconf
- En premier lieu dans dans le répertoire racine ( si on dispose d'un répertoire racine) on créé les fichiers **Makefile.am** nécessaires à automake. Ces fichiers renseignent uniquement autoconf sur les sources du programme. Le fichier Makefile.am et le fichier configure.ac, via la commande aclocal généreront un fichier aclocal.m4 compatible avec la macro m4

```
SUBDIRS=src
```

- Puis dans les répertoires concernés comme par exemple src

```
bin_PROGRAMS=exemple
exemple_SOURCES=   main.c   afficher.c afficher.h
```

# Modification de configure.ac

- Modifier configure.ac ( ne fonctionne pas avec C++)

```
# version d'autoconf (AC=autoconf, AM=automake)
AC_PREREQ([2.69])
#Initialisation pas de blanc avant les parentheses
AC_INIT(Exemple autotools, 1.0, gilles@gillesmaire.com)
AM_INIT_AUTOMAKE
AC_PROG_CC
AC_PROG_MAKE_SET
# fichiers Make
AC_CONFIG_FILES([
 Makefile
 src/Makefile
  ])
# Derniere instruction
AC_OUTPUT
```

# Configure.ac pour C++

```
AC_PREREQ([2.69])
AC_INIT(Exemple autotools, 1.0, gilles@gillesmaire.com)
AM_INIT_AUTOMAKE
# AM : automake AC : autoconf
AM_PROG_RANLIB
# Libraries
AC_LANG(C++)
AC_PROG_CXX
AC_CONFIG_FILES([
Makefile
src/Makefile
])
AC_OUTPUT
```

# Génération automake

- lancer :

```
aclocal
autoconf
automake -a -c
```

- messages d'erreur car il manque les fichiers NEWS (nouveautés), README (infos), AUTHORS (les auteurs), ChangeLog (suivi des modifications)
- créer ces fichiers dans la racine

```
automake -a -c
./configure
make
```

# Ajouter des sources

- On modifie Makefile.am
- on relance `automake` ou `make`
- pour ajouter le support d'une librairie :
  - dans configure.ac
    - ajouter AC_SEARCH_LIBS(pow, m)
  - relancer aclocal, autoconf et automake -c -a
- `make dist` crée le fichier tar de source
- `make distclean` ne laisse que les fichiers nécessaires au configure

# Autoscan

- La commande autoscan peut être employée pour tenter de fabriquer un fichier configure.ac via un fichier conficgure.scan
- fabrication de la structure des sources :

```
mkdir rc
mkdir rc/include
autoscan
# Vérification du fichier configure.scan
mv configure.scan configure.ac
```

# CMake

## Rubriques

- Premiers pas
- Options

# Premiers pas

# Présentation

- CMake est un système de construction logicielle.
  - logiciel libre (licence BSD)
  - multilangage (C, C++, Java)
  - multiplateforme (Windows, Linux, MacOsX)
- Le fichier de description est **CMakeLists.txt**
  - décrit la configuration logicielle
  - il permet la génération des fichiers Makefile, du fichier projet de l'environnement de développement
- Un fichier CMakeCache.txt est dédié à chaque utilisateur
- CMake est fourni avec une interface graphique
- Le site officiel de CMake est https://cmake.org/

# Installation

- Pour Windows on trouve des installateurs MSI ou Zip à l'adresse https://cmake.org/install/
- Pour Ubuntu, Debian

```
sudo apt-get install cmake
```

- Pour RedHat, Fedora et dérivés

```
dnf install cmake
```

- Pour ArchLinux et dérivé

```
pacman -S cmake
```

# Syntaxe fichie CMakeLists.txt

- Chaque entrée du fichier est constituée d'une commande et d'arguments sur une ou plusieurs lignes encadrés par des parenthèses comme suit :

```
(commande argument1 argument2)
```

ou

```
(commande
    argument1
    argument2)
```

- Le séparateur d'argument est donc le caractère espace ou le retour ligne
- Par contre entre la parenthèse et le nom de la commande aucun retour ligne n'est admis.
- Si un argument contient des caractères blancs, on pourra encadrer l'argument de doubles quotes
- Enfin les commentaires sont les lignes commençant par #

# Premier fichier CMakeLists.txt

- **project (NomDuProjet)** : permet de déclarer le projet
- **project (NomDuProjet C++)**: permet de déclarer le projet ainsi que le langage si vous utilisez des extensions non standards à vos fichiers de développement
- **add_executable(**`nomexecutable fichiersource1 fichiersource2 ...`**)** : indique le nom du fichier exécutable (sans .exe) et la liste des fichiers sources
- Les répertoires sont notés / que vous soyez sous Windows ou Unix
- Exemple

```
#Déclaration du projet
project(MyProject)
#Déclaration de l'exécutable
add_executable(
        my_executable
        src/classe.h
        src/clase.cpp
        src/main.cpp )
```

# Lancement de cmake

- La syntaxe est :

```
cmake repertoire -G"Nom du générateur"
```

- On indique par cette commande le nom du répertoire où est contenu le fichier CMakeLists.txt
- Le deuxième argument est le nom du générateur, par défaut on génèrera un Makefile mais la commnande suivante donne la liste des générateurs

```
cmake --help
```

- Ainsi les appels suivants sont valides :

```
cmake . -G"Unix Makefiles"
cmake ..-G"CodeBlocks - Unix Makefiles"
```

- Les fichiers résultat seront générés à l'endroit où vous appelez cmake

# Options

# Commande file

- **cmake_minimum_required(VERSION** 2.8**)** : indique la version minimum de cmake supportée
- **file (**`param1 param1 param3`**)**: permet de lister les fichiers sans les nommer expressément,
  - param1 peut êre GLOB ou GLOB_RECURSE
  - param2 est le nom de la variable affectée qui sera utilisée sous la forme ${variable}
  - param3 est une expression si param1 est GLOB ou un répertoire si param1 est GLOB_RECURSE (les sous dossiers du dossier spécifié)

```
file(
        GLOB_RECURSE
        source_files
        src/*
)
add_executable (my_executable ${source_files})
```

- cmake devra être appelé à chaque fois qu'on ajoute un fichier afin que la variable source_files soit mise jour

# CMakeCache.txt

- On a mis dans CMakeLists.txt la liste des informations invariantes
- On va mettre dans CMakeCache.txt la listes des informations qui varient d'une compilation à l'autre (mode debug) ou d'une machine à l'autre (nom du répertoire source ou chemin des compilateurs)
- Après avoir fait un cmake sur un fichier CMakeLists.txt on verra que le fichier CMakeCache.txt est créé et qu'il contient des variables de la forme

```
NOM_DE_LA_VARIABLE:TYPE=valeur
```

- Par exemple :

```
CMAKE_CXX_COMPILER:FILEPATH=/usr/bin/c++
CMAKE_CXX_FLAGS_DEBUG:STRING=-g
CMAKE_C_FLAGS_RELEASE:STRING=-O3 -DNDEBUG
CMAKE_BUILD_TYPE:STRING=Release
```

- On peut également court-circuiter l'utilisation du fichier CMakeCache.txt en positionnant la variable via l'option -D de cmake

```
cmake . -DCMAKE_BUILD_TYPE:STRING=Release
```

# FAQ

- Question : comment supprimer les fichiers générés comme l'exécutable ?
  - Réponse :

```
make clean
```

- Question : comment supprimer les fichiers générés CMakeFiles CMakeCache.txt et autre ?
  - Réponse : en général on ne positionne pas ces fichiers dans le même répertoire que les sources et on les supprime en supprimant le répertoire généré, c'est la raison pour laquelle on ne trouve pas de commande spécifiques.
- Question : comment compile-t-on en mode debug ?
  - Réponse : on crée un répertoire Debug dans le quel on lance cmake

```
mkdir debug
cd debug
cmake ../src -DCMAKE_BUILD_TYPE:STRING=Debug
make
```