

# Makefile

Anne Cadiou

Laboratoire de Mécanique des Fluides et d'Acoustique

Ateliers et séminaires pour l'informatique et le calcul scientifique  
PMCS2I - LMFA  
Vendredi 7 septembre 2018



## Définition

- `Makefile` est un fichier texte, utilisé par l'utilitaire `make` (datant des années 1970. partie intégrante d'Unix.).
- Le fichier `Makefile` regroupe une série d'instructions et de règles permettant d'exécuter des actions.
- Utilisé notamment dans la compilation, la construction et la maintenance de projets incluant des tâches complexes avec dépendances (codes de calcul, documents  $\text{\LaTeX}$ , etc.).
- Implémentation la plus utilisée : `make` du projet GNU, disponible sur la majorité des systèmes Linux.

Un `Makefile` est un descripteur de l'utilitaire `make`.

## Génération

Le fichier Makefile peut être généré

- manuellement
- ou
- avec des outils de construction de projets.

Outils généraux de construction (et packaging)

CMake, Autotools, Ant, Scons, etc.

suivent une démarche connue :

```
./configure; make; make install
```

Ici on se focalise sur la syntaxe du fichier Makefile pour la construction de codes de calcul.

## Comment ça marche ?

Lorsque la commande `make` est exécutée, elle cherche par défaut en entrée les fichiers nommés `makefile`, `Makefile` ou `GNUmakefile`

`make` détermine automatiquement quels fichiers doivent être recompilés et exécute les commandes nécessaires.

Le `Makefile` contient les **règles** de compilation, d'édition de lien et de dépendances, par des macros et des règles implicites.

## À quoi ça ressemble ?

```
CC          = g++
CLINKER     = g++

OBJS        = prog.o

.SUFFIXES: .o .cpp
.cpp.o:
    $(CC) -c $(CCLAGS) $<

EXE = a.out

all : $(EXE)

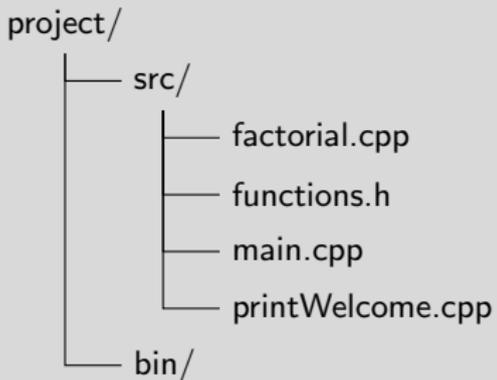
$(EXE): $(OBJS)
    $(CLINKER) $(LDFLAGS) -o $@ $(OBJS)

clean:
    rm -f $(OBJS) $(EXE)

install:
    mv $(EXE) ../bin/.
```

## Exemple simple

### Structure du projet



## Programmes

```
/**
2  * @file    main.cpp
3  * @author  Anne Cadiou from Zhiliang Xu, Notre Dame Univ.
4  * @brief   example for Makefile tutorial
5  */
6  #include <iostream>
7  #include "functions.h"
8
9  using namespace std;
10
11 int main()
12 {
13     printWelcome();
14     cout << "\nThe factorial of 5 is " << factorial(5) << endl;
15     return 0;
16 }
```

```
/* functions.h */
2  #ifndef _FUNC_H_
3  #define _FUNC_H_
4     void printWelcome();
5     int factorial(int n);
6  #endif
```

## Fonctions

```
/* printWelcome.cpp */
2 #include <iostream>
  #include "functions.h"
4
void printWelcome() { std::cout << "Welcome!" << std::endl; }
```

```
/* factorial.cpp */
2 #include "functions.h"
4 int factorial(int n)
  {
6   int i, fac = 1;
   if (n != 1) {
8     for (i=1; i<= n; i++)
       fac *= i;
10    return fac;
   }
12   else
     return 1;
14 }
```

## Sans Makefile

### Lignes de commandes

```
acadiou@plume:~/projet/src$ g++ -c main.cpp printWelcome.cpp
    factorial.cpp functions.h
acadiou@plume:~/projet/src$ g++ -o exe main.o printWelcome.o
    factorial.o
```

```
acadiou@plume:~/projet/src$ mv exe ../bin/.
```

### Exécution

```
acadiou@plume:~/projet$ ./bin/exe
Welcome!

The factorial of 5 is 120
```

Toute modification de l'un des programmes nécessite de le recompiler séparément et faire à nouveau l'édition de liens.

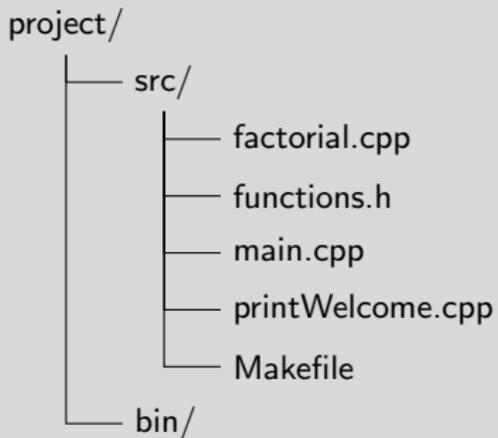
```
acadiou@plume:~/projet/src$ g++ -c main.cpp
acadiou@plume:~/projet/src$ g++ -o exe main.o printWelcome.o
    factorial.o
```

## Pourquoi ne pas se contenter d'appeler le compilateur ?

Dans les codes de calculs, la construction de l'exécutible est une tâche parfois délicate. Elle dépend

- d'un **grand nombre de fichiers**
- avec de nombreuses **dépendances**
- du cas à traiter (compilation **conditionnelle**, ...)
- de nombreux **éléments externes** (bibliothèques, ...)
- et de la manière dont ils sont installés
- de la **plateforme**
- de l'**environnement** sur le plateforme

## Création manuelle d'un Makefile



## Syntaxe de base d'un Makefile

Le Makefile est constitué de **règles** sous la forme suivante :

```
# COMMENT CIBLE
CIBLE: DEPENDANCES
<TAB>  COMMANDE
<TAB>  COMMANDE
<TAB>  COMMANDE

CIBLE: DEPENDANCES
<TAB>  COMMANDE
<TAB>  COMMANDE
<TAB>  COMMANDE
```

- Ces règles peuvent être explicites ou implicites.
- La liste des dépendances est séparée d'un espace ou affichée sur plusieurs lignes.
- Attention, bien respecter les tabulations précédant chaque ligne de commande.

## Fonctionnement de make

- `make` ne recompile que ce qui a été modifié.
- `make` regarde si les dépendances sont satisfaites :
  - Si elles ne le sont pas, prend la première dépendance pour cible.
  - Si elles le sont, exécute les commandes associées à la règle.
- En terme d'exécution, on peut voir `make` comme un programme récursif.

Exemple :

```
# Principal rule
exe : main.o
    g++ main.o -o exe

# Explicit rule
main.o : include.h main.cpp
    g++ -c main.cpp

# Other rule
clean :
    rm main.o
```

## Makefile naïf sur le projet

```
exe: \  
    printWelcome.o \  
    factorial.o \  
    main.o  
    g++ -o exe printWelcome.o factorial.o main.o  
  
printWelcome.o: printWelcome.cpp functions.h  
    g++ -o printWelcome.o -c printWelcome.cpp  
  
factorial.o: factorial.cpp functions.h  
    g++ -o factorial.o -c factorial.cpp  
  
main.o: main.cpp functions.h  
    g++ -o main.o -c main.cpp  
  
install:  
    mv exe ../bin/.  
  
clean:  
    rm -f *.o
```

## Utilisation

Pour savoir ce que va faire la commande make sans l'exécuter :

```
cadiou@plume:~/project/src# make -n
g++ -o printWelcome.o -c printWelcome.cpp
g++ -o factorial.o -c factorial.cpp
g++ -o main.o -c main.cpp
g++ -o exe printWelcome.o factorial.o main.o
```

```
cadiou@plume:~/project/src# ls -lrt
total 20
-rw-rw-r-- 1 acadiou acadiou 107 juil 30 15:13 functions.h
-rw-rw-r-- 1 acadiou acadiou 197 juil 30 15:16 factorial.cpp
-rw-rw-r-- 1 acadiou acadiou 194 juil 30 18:15 main.cpp
-rw-rw-r-- 1 acadiou acadiou 232 juil 30 18:41 printWelcome.cpp
-rw-rw-r-- 1 acadiou acadiou 371 juil 30 19:17 Makefile
```

Exécution de la cible par défaut de la cible install :

```
cadiou@plume:~/project/src# make
cadiou@plume:~/project/src# make install
mv exe ../bin/
cadiou@plume:~/project/src# cd ../bin
cadiou@plume:~/project/bin# ./exe
```

## Principales options de make

Tester sans exécuter :

```
make -n
```

Lire un fichier nommé filename (i.e. autre que makefile, Makefile ou GNUmakefile) :

```
make -f filename
```

Réaliser la compilation sur nprocs processus :

```
make -j nprocs
```

Écrire sur la sortie standard le règles, variables et environnement du Makefile :

```
make -p
```

Exécuter les commandes associées à la cible CIBLE :

```
make CIBLE
```

Par défaut la cible est celle qui est construite. Les autres cibles ne sont traitées que si elles sont des dépendances de la cible.

## Cibles classiques

- all** : génération de tous les exécutable.
- .PHONY** : les dépendances sont toujours reconstruites.
- clean** : suppression des fichiers intermédiaires.
- mrproper** : suppression des fichiers intermédiaires et des exécutable.

## Application à l'exemple

```
all: exe

exe: \
    printWelcome.o \
    factorial.o \
    main.o
    g++ -o exe printWelcome.o factorial.o main.o

printWelcome.o: printWelcome.cpp functions.h
    g++ -o printWelcome.o -c printWelcome.cpp

factorial.o: factorial.cpp functions.h
    g++ -o factorial.o -c factorial.cpp

main.o: main.cpp functions.h
    g++ -o main.o -c main.cpp

install:
    mv exe ../bin/

clean:
    rm -f *.o core

mrproper: clean
    rm -f exe
```

## Variables

- déclaration : `NOM = VALEUR`
- utilisation : `$(NOM)`

Variables prédéfinies :

- CFLAGS** : options de compilation pour le C
- CPPFLAGS** : options de précompilation
- CXXFLAGS** : options de compilation pour le C++
- FCFLAGS** : options de compilation pour le FORTRAN
- CC, CXX ou FC** : désignent les compilateurs (C, C++, FORTRAN)
- LDFLAGS** : options d'édition de liens

Exemple d'utilisation (réaffectation)

```
CC = gcc
CFLAGS = -Ofast -Wall
LD = gcc
LDFLAGS = -s
```

Lorsqu'on ajoute un élément à une variable, on utilise le `+=` afin d'éviter toute récursivité.

```
FCFLAGS = -ffixed-line-length-none
FCFLAGS += -I/softs/software/OpenMPI/1.10.2-GCC-4.9.3-2.25/include
```

## Prise en compte dans le Makefile

```
1 CXX=g++
2 CXXFLAGS=
3 LDFLAGS=
4 EXE=exe
5
6 all: $(EXE)
7
8 $(EXE): \
9     printWelcome.o \
10    factorial.o \
11    main.o
12     $(CXX) -o $(EXE) printWelcome.o factorial.o main.o $(LDFLAGS)
13
14 printWelcome.o: printWelcome.cpp functions.h
15     $(CXX) $(CXXFLAGS) -o printWelcome.o -c printWelcome.cpp
16
17 factorial.o: factorial.cpp functions.h
18     $(CXX) $(CXXFLAGS) -o factorial.o -c factorial.cpp
19
20 main.o: main.cpp functions.h
21     $(CXX) $(CXXFLAGS) -o main.o -c main.cpp
22
23 install:
24     mv $(EXE) ../bin/.
```

```
26 .PHONY: all clean mrproper
28
30 clean:
    rm -f *.o core
32 mrproper: clean
    rm -f $(EXE)
```

## Utilisation

Pour nettoyer l'ensemble du projet :

```
cadiou@plume:~/project/src# make mrproper
rm -f *.o core
rm -f exe
```

Pour modifier une variable à l'exécution :

```
acadiou@plume:~/projet/src# make CXX=c++
c++ -o printWelcome.o -c printWelcome.cpp
c++ -o factorial.o -c factorial.cpp
c++ -o main.o -c main.cpp
c++ -o exe printWelcome.o factorial.o main.o
```

ou encore :

```
acadiou@plume:~/projet/src# make CXXFLAGS="-Wall -O3"
g++ -o printWelcome.o -c printWelcome.cpp -Wall -O3
g++ -o factorial.o -c factorial.cpp -Wall -O3
g++ -o main.o -c main.cpp -Wall -O3
g++ -o exe printWelcome.o factorial.o main.o
```

## Variables internes

- `$@` représente la cible.
- `$$` représente la liste des dépendances.
- `$<` représente la première dépendance.
- `$$?` représente la liste des dépendances plus récentes que la cible.
- `$$*` représente le nom de la cible sans suffixe.

Exemple :

```
# $$ == rk4 et $$ == main.o rk4.o
rk4: main.o rk4.o
    $(CC) -o $$ $$
main.o: main.cpp rk4.h
    $(CC) -o $$ -c $<
rk4.o: rk4.cpp rk4.h
    $(CC) -o $$ -c $<
```

## Prise en compte dans le Makefile

```
1 CXX=g++
2 CXXFLAGS=
3 LDFLAGS=
4 EXE=exe
5
6 all: $(EXE)
7
8 $(EXE): \
9     printWelcome.o \
10    factorial.o \
11    main.o
12     $(CXX) -o $@ $^ $(LDFLAGS)
13
14 printWelcome.o: printWelcome.cpp functions.h
15     $(CXX) $(CXXFLAGS) -o $@ -c $<
16
17 factorial.o: factorial.cpp functions.h
18     $(CXX) $(CXXFLAGS) -o $@ -c $<
19
20 main.o: main.cpp functions.h
21     $(CXX) $(CXXFLAGS) -o $@ -c $<
22
23 install:
24     mv $(EXE) ../bin/.
```

## Règles d'inférence

Ce sont de règles génériques. Dans l'exemple, les règles dont les fichiers objets sont les cibles sont identiques. Pour les définir, on utilise des macros ou règles formelles (utilisant le caractère %).

Exemple :

```
rk4: main.o rk4.o
    $(CXX) -o $@ $^ $(LDFLAGS)

%.o :%.cpp
    $(CXX) -o $@ -c $< $(CXXFLAGS)

main.o: rk4.h
rk4.o: rk4.h
```

## Prise en compte dans le Makefile

```
2 CXX=g++
LDFLAGS=
4 EXE=exe

6 all: $(EXE)

8 $(EXE): \
    printWelcome.o \
10    factorial.o \
    main.o
12    $(CXX) -o $@ $^ $(LDFLAGS)

14 printWelcome.o: fonctions.h
factorial.o: fonctions.h
16 main.o: fonctions.h

18 %.o: %.cpp
    $(CXX) $(CXXFLAGS) -o $@ -c $<

20
22 install:
    mv $(EXE) ../bin/.
```

## Manipulation des noms des fichiers source

**wildcard** : récupération des noms de fichiers

```
SRC=$(wildcard *.cpp)
```

**patsubst** : remplacement d'une expression par une autre

```
OBJ=$(SRC:.cpp=.o)  
OBJ=$(patsubst %.cpp,%.o,$(SRC))
```

**notdir** : extraction du nom du fichier

```
OBJ=$(notdir \$(patsubst %.cpp,%.o,$(SRC)))
```

## Application sur la liste des fichiers

```
1 CXX=g++
2 CXXFLAGS=
3 LDFLAGS=
4 EXE=exe
5 SRC=$(wildcard *.cpp)
6 OBJ=$(SRC:.cpp=.o)
7
8 all: $(EXE)
9
10 $(EXE): $(OBJ)
11     $(CXX) -o $@ $^ $(LDFLAGS)
12
13 %.o: %.cpp
14     $(CXX) $(CXXFLAGS) -o $@ -c $<
15
16 printWelcome.o: functions.h
17 factorial.o: functions.h
18 main.o: functions.h
19
20 install:
21     mv $(EXE) ../bin/.
```

```
22 .PHONY: all clean mrproper
24
26 clean:
    rm -f *.o core
28 mrproper: clean
    rm -f $(EXE)
```

## Dépendances

Dans l'exemple, seules les dépendances sont explicitées. Pour définir automatiquement les dépendances, on peut utiliser `makedepend`.

```
makedepend printWelcome.cpp factorial.cpp main.cpp
```

ajoute à la fin du fichier Makefile les dépendances qu'il a trouvées :

```
# DO NOT DELETE

factorial.o: functions.h
printWelcome.o: functions.h
main.o: functions.h
```

La règle correspondante serait alors :

```
depend:
    makedepend -- $(CXXFLAGS) -- $(SRC)
```

`makedepend` a initialement été développé pour le C. En C++ avec `g++`, utiliser l'option `-M` pour écrire les dépendances dans un fichier à concaténer au Makefile

```
depend: $(SRC) $@
    $(CXX) -M $^ >$@

-include depend
```

# Application

```
2 CXX=g++
LDFLAGS=
4 EXE=exe
SRC=$(wildcard *.cpp)
6 OBJ=$(SRC:.cpp=.o)

8 all: $(EXE)

10 $(EXE): $(OBJ)
    $(CXX) -o $@ $^ $(LDFLAGS)

12 %.o: %.cpp
14    $(CXX) $(CXXFLAGS) -o $@ -c $<

16 install:
    mv $(EXE) ../bin/.
```

```
18 .PHONY: all clean mrproper
20
22 clean:
    rm -f *.o core
24 mrproper: clean
    rm -f $(EXE)
26
28 depend: $(SRC) $@
    $(CXX) -M $^ >$@
30 -include depend
```

## Parties conditionnelles

Il est possible de n'exécuter que certaines parties du `Makefile` sous certaines conditions. Cela permet par exemple de définir les conditions de compilation en mode `DEBUG` ou les options suivant différentes `MACHINES` ou différentes versions de compilateurs, versions de bibliothèques, suivant l'environnement de l'utilisateur, etc.

Syntaxe de la compilation conditionnelle :

- Teste l'égalité des arguments

```
ifeq(arg1, arg2)
```

- Teste l'inégalité des arguments

```
ifneq(arg1, arg2)
```

- Teste si la variable existe (est non vide)

```
ifdef var
```

- Teste si la variable n'existe pas (est vide)

```
ifndef var
```

## Utilisation de conditions

### Exemple pour spécifier une compilation en mode DEBUG

```
CXXFLAGS=-Wall -ansi
LDFLAGS=-Wall -ansi

ifdef MODE_DEBUG
    CXXFLAGS+= -g
    LDFLAGS+= -g
endif
```

### Exemple pour modifier les variables suivant la machine utilisée

```
ifeq ($(MACHINE),p2chpd_ompi)
FC = mpif77
FCFLAGS = -ffixed-line-length-none
FCFLAGS += -I/softs/mpi/openmpi/1.8.4_gcc47/include
LDFLAGS = -lmpi
else ifeq ($(MACHINE),newton_ompi)
FC = mpif77
FCFLAGS = -ffixed-line-length-none
FCFLAGS += -I/softs/software/OpenMPI/1.10.2-GCC-4.9.3-2.25/include
LDFLAGS = -lmpi
else
FC = mpif77
FCFLAGS = -ffixed-line-length-none
FCFLAGS += -I/usr/lib/openmpi/include
LDFLAGS = -lmpi
endif
```

## Application dans l'exemple

```
1 CXX=g++
2 CXXFLAGS=
  LDFLAGS=
4 EXE=exe
  SRC=$(wildcard *.cpp)
6 OBJ=$(SRC:.cpp=.o)

8 ifeq ($(DEBUG),True)
  CXXFLAGS = -g
10 else
  CXXFLAGS += -O3 -Wall
12 endif

14 all: $(EXE)

16 $(EXE): $(OBJ)
  $(CXX) -o $@ $^ $(LDFLAGS)

18 %.o: %.cpp
20   $(CXX) $(CXXFLAGS) -o $@ -c $<

22 install:
  mv $(EXE) ../bin/.
```

S'utilise par

```
acadiou@plume:~/projet/src# make DEBUG=True
```

## Autres macros

Les macros peuvent servir à automatiser encore de nombreuses tâches, comme établir la liste des fichiers objets, insérer un git SHA1 dans l'exécutable, etc.

Des fonctions permettent également de descendre dans des sous-répertoires, d'inclure des morceaux de Makefile génériques, etc.

```
include Makefile.global

all: $(OBJ)
    for i in $(SUBDIRS); do (cd $$i; $(MAKE) all); done
```

## Faire des tests

Dans le Makefile, les cibles peuvent servir à définir des tests unitaires.

Par exemple :

```
test: test1 test2

test1:
    exe < input1.txt > output.txt
    diff correct1.txt output.txt

test2:
    exe < input2.txt > output.txt
    diff correct2.txt output.txt
```

## Remarques et références

- automatise le compilation des fichiers
- `make` et `Makefile` intègrent de nombreuses fonctions, pas toutes exposées ici (boucles, fonctions, macros, etc.)
- le fichier `Makefile` contient une liste de règles et de dépendances utilisées pour construire des cibles
- chaque commande est précédée d'une tabulation
- `make` est Turing complet
- Documentation :  
<http://www.gnu.org/software/make/manual/>  
<http://clarkgrubb.com/makefile-style-guide>