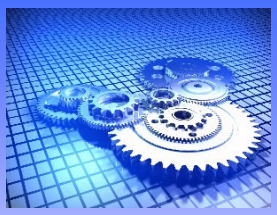


Les outils de construction de programmes

Plan

- Introduction
- Rappels
- Makefile
- Autotools
- CMake
- Scons

Introduction

Problématique

- Transformer un code source en exécutable
  - Sous Unix/Linux make
  - IDE (KDevelopp, Microsoft Visual Studio, Eclipse)
- Disparités selon système d'exploitation ou matériel
  - Dépendances (bibliothèques, programmes)
  - Compilateur et options
- Plusieurs makefiles ?
  - Makefiles paramétrables
    - Autotools
      - Combinaison d'outils, plusieurs syntaxes, plusieurs commandes
      - Bonne gestion du matériel mais système dépendant
- Outils indépendants du système
  - S'appuient indirectement sur des outils de construction propres à chaque système / IDE
  - Multi-plateformes et transparents pour l'utilisateur
  - Cmake, Scons
    - Un seul outil, un seul langage, facilement paramétrable

Remarques

- Langages utilisés C et C++
- Exemples exécutés sous Linux
- Compilateur utilisé gcc et g++, ainsi que GNU make
- On ne parlera pas ici des outils plus spécifiques au monde Java
  - Ant
  - JAM

Rappels

Bonnes pratiques

- Organiser le code source en plusieurs fichiers
  - Des fichiers contenant le corps des fonctions et procédures.
    - d'extension .c ou .cpp
  - Des fichiers contenant :
    - Déclarations des prototypes des fonctions et procédures
    - Déclarations des structures de données communes, d'extension .h
- Augmente la lisibilité du code
- Permet de réutiliser et de mutualiser du code
  - Générer des bibliothèques

Bonnes pratiques

Exemple : gestion d'une pile

// fichier pile.h

```
#ifndef DEFINITION
#define DEFINITION

#define NB_PILE 5

Typedef Pile {…};

Pile Empiler(Pile, int);
Pile Depiler(Pile);
Int Parcourir(Pile, int);

#endif
```

// fichier pile.c

```
#include "pile.h"

Pile Empiler(Pile, int) {…}

Pile Depiler(Pile) {…}

Int Parcourir(Pile, int) {…}
```

// fichier principal.c

```
#include "pile.h"

int main(int argc, char *argv[])
{
        P Pile; int i;
        for (i=1, i<=argc, i++)
        {
                Empiler(P,atoi(argv[i]));
        }
…
}
```

Bonnes pratiques

- Isoler le code de l'application des instructions de débogage

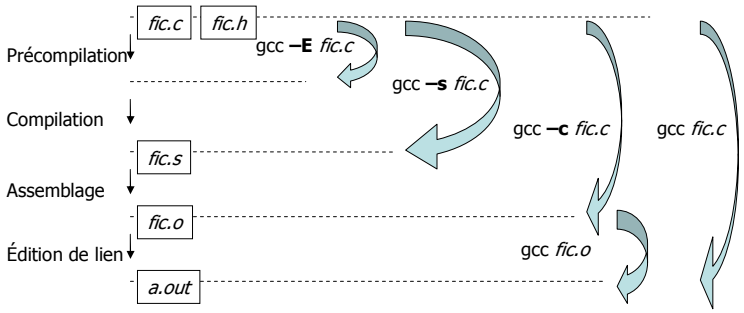
```
#ifdef DEBUG
    Printf("la valeur de  …) ;
#endif
```

- Pas visible par défaut
- Peut être activé lors de la précompilation
  - gcc -DDEBUG=1 fichier.c

Compilation avec gcc

- Ensemble d'outils : préprocesseur, compilateur, assembleur, linker
- Préprocesseur (cpp)
  - Supprime les commentaires, effectue les inclusions, évalue les macros,… (traite les lignes débutant par un #)
- Compilateur (cc1)
  - Converti le code C ou C++ en assembleur
- Assembleur (as)
  - Assemblage des fichiers assembleurs, génère les fichiers objets
- Éditeur de lien (ld)
  - Détermine les adresses mémoires des symboles et lie les fichiers objets entre eux

Compilation C avec gcc



Exemple

- gcc -c pile.c principal.c pile.h ;
  gcc -o prog pile.o principal.o ;
- gcc pile.c principal.c

Les librairies

- Archives de fonctions mutualisées
  - Généralement dans /usr/lib/ ou /lib/
- Statique
  - lib*xxx*.a
  - ar -rv lib*xxx*.a fic1.o fic2.o … ficn.o
- Dynamique
  - lib*xxx*.so
  - gcc -o lib*xxx*.so -shared fic1.o fic2.o … ficn.o
  - Renseigner *LD_LIBRARY_PATH, ou /etc/ld.so.conf et utiliser ldconfig*
- Statique vs dynamique
  - Statique
    - code exécutable de la librairie dans le fichier exécutable final
  - Dynamique
    - Chargement dynamique du code exécutable de la librairie lors de l'exécution du programme

Quelques options utiles

- gcc -Wall *fichier.c*
  - Active tous les warning possibles
- gcc *fichier1.o fichier2.o* -l*nom* :
  - Lie si la librairie *libnom.so ou libnom.a* au code exécutable
- gcc -L*répertoire fichier1.o fichier2.o*
  - Ajoute répertoire au chemin de recherche des librairies
- gcc –Dmacro=def fichier.c
  - Fixe à def la valeur de la macro dans le fichier source

Quelques options utiles

- gcc -g *fichier.c*
  - Génère les informations symboliques de débogage
  - Permet alors d'utiliser « gdb » ou « valgring » pour déboguer
- gcc -o *nom fichier.c*
  - Nomme « nom » le fichier obtenu en résultat de l'exécution de gcc
- gcc -I*répertoire fichier.c*
  - Ajoute « répertoire » au chemin de recherche des fichiers d'entêtes (par défaut /usr/include/)

Makefile

Généralités

- Automatise la compilation des fichiers
- Exécution de commandes shell
- N'exécute que les commandes nécessaires
- Basé sur un fichier Makefile et l'appel à la fonction make

Fichier Makefile

- Composé de plusieurs règles de la forme :

cible : *dépendances*
<TAB> *commandes*
……
cible : *dépendances*
<TAB> *commandes*

- Chaque commande est précédée d'une tabulation

Exemple :

```
prog: principal.o pile.o
    gcc -o prog pile.o principal.o
principal.o : principal.c pile.h
    gcc –c principal.c
pile.o : pile.c pile.h
    gcc –c pile.c
```

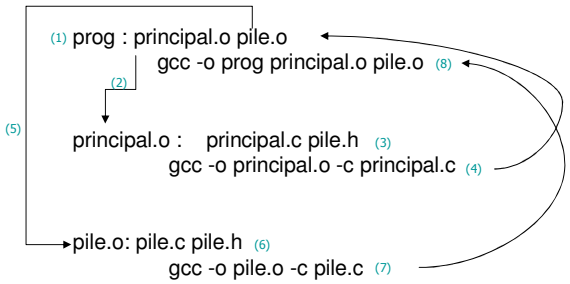

Évaluation d'une règle

- Lors de l'utilisation de la commande « make *cible* »
  - Évaluation de la règle dont la cible est passée en argument à make
  - Évaluation de la première règle rencontrée (s'il n'y a pas d'argument)
- Les dépendances sont analysées
  - Si une dépendance n'est pas à jour
    - Si cette dépendance est cible d'une autre règle du Makefile
      - cette règle est à son tour évaluée
  - Lorsque l'ensemble des dépendances sont à jour
    - Si la cible n'existe pas
    - Si une de ses dépendances est plus récente qu'elle,
      - Les commandes de la règle sont exécutées

Évaluation d'une règle

Exemple :

make prog



Cibles classiques

- all
  - Génération de tous les exécutables
- .PHONY
  - Dépendances sont toujours régénérées
- clean
  - Suppression des fichiers intermédiaires
- Mrproper
  - Suppression des exécutables
  - Suppression des fichiers intermédiares

Exemple

```
all: prog
prog : principal.o pile.o
	gcc -o prog principal.o pile.o
principal.o: principal.c
	gcc -o principal.o -c principal.c
pile.o: pile.c pile.h
	gcc -o pile.o -c pile.c
.PHONY: clean mrproper
clean:
	rm -rf *.o
mrproper: clean
	rm -rf prog
```

Variables personnalisées

- Déclaration *NOM=VALEUR*
- Utilisation *$(NOM)*
- CFLAGS
  - Options de compilation pour le C
- CPPFLAGS
  - Options de précompilation
- LDFLAGS
  - Options d'édition de liens
- CXXFLAGS
  - Options de compilation pour le C++
- CC ou CXX pour les compilateurs

Exemple

```
CC=gcc
CFLAGS=-Wall
LDFLAGS=-Wall –lm
EXEC=prog

all: $(EXEC)

prog : principal.o pile.o
    $(CC) -o prog principal.o  \
                $(LDFLAGS)
pricipal.o: principal.c
    $(CC) -o principal.o -c principal.c \
                $(CFLAGS)
…
```


Variables internes

- $@
  - La cible
- $<
  - La première dépendance
- $^
  - La liste des dépendances
- $?
  - La liste des dépendances plus récentes que la cible
- $*
  - Le nom de la cible sans suffixe

Exemple :

```
prog : principal.o pile.o
    $(CC) -o $@ $^

principal.o : principal.c pile.h
    $(CC) -o $@ -c $<

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

Règles d'inférences

- %.o: %.c
  commandes
- .c.o:
  commandes

Exemple :

```
prog : principal.o pile.o
    $(CC)  -o  $@  $^  $(LDFLAGS)

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


Manipulation des noms de fichiers sources

- wildcard
  - Récupération de noms de fichiers

    Exemple : SRC= $(wildcard src/*.c)
- patsubst
  - Remplacement d'une expression par une autre

    Exemple : OBJ= $(SRC:.c=.o)
    ou OBJ= $(patsubst %.c,%.o,$(SRC))
- notdir
  - Extraction du nom de fichier

    Exemple :OBJ= $(notdir $(patsubst %.c,%.o,$(SRC)))

Utilisation de conditions

Exemple :

```
MODE_DEBUG=no

ifeq ($(MODE_DEBUG),yes)
        CFLAGS=-Wall -ansi -g
        LDFLAGS=-Wall -ansi -g
else
        CFLAGS=-Wall -ansi
        LDFLAGS=-Wall -ansi
endif
```

- Appeler make en mode déboguage
  - make MODE_DEBUG=yes

Recherche de dépendances

- En utilisant la variable VPATH
  - Modifie les chemins de recherche

    Exemple : VPATH = src:../headers
    pile.o : pile.c

  - pile.c sera recherché dans le répertoire courant ainsi que dans le répertoire headers et dans le répertoire src du répertoire courant

- En utilisant la directive vpath
  - Chemin de recherche pour une classe de fichiers

    Exemple : vpath %.h ../headers

  - Les fichiers d'entête seront recherchés dans le répertoire headers du répertoire courant

Appel imbriqué de makefile

Makefile appelant (dans répertoire racine)

```
export CC=gcc
export CFLAGS=-Wall
export LDFLAGS=-Wall
SRC_DIR=src
EXEC=$(SRC_DIR)/prog

all: $(EXEC)

$(EXEC):
	cd $(SRC_DIR) && $(MAKE)

.PHONY: clean mrproper $(EXEC)

clean:
	cd $(SRC_DIR) && $(MAKE) $@

mrproper: clean
	cd $(SRC_DIR) && $(MAKE) $@
```

Makefile appelé (dans répertoire src)

```
EXEC=prog
SRC= $(wildcard *.c)
OBJ= $(SRC:.c=.o)

all: $(EXEC)

prog : $(OBJ)
	$(CC) -o $@ $^ $(LDFLAGS)

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

.PHONY: clean mrproper
clean:
	rm -rf *.o
mrproper: clean
	rm -rf $(EXEC)
```


Autotools

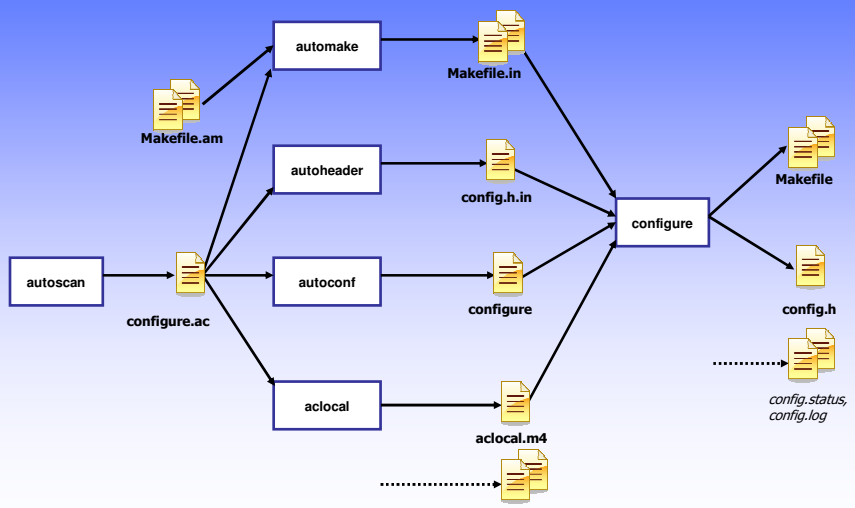
Généralités

- Génération automatique de makefile
  - Makefiles génériques, paramétrables
  - Configurés automatiquement selon la plateforme
- Portabilité des développements
  - Multi plateforme
- Construction simple et transparente pour l'utilisateur
  - ./configure, make, make install
  - FHS Unix
  - Paramétrable par l'utilisateur

Généralités

- Problématique datant de 1991
  - Cygnus, Imake, Autoconf, GccConfigure, MetaConfig
  - Établir la configuration de la machine et du système d'accueil
  - Construire un Makefile à partir de celle-ci
- Un ensemble d'outils
  - Automake
    - Génération de makefiles génériques et portables
  - Autoconf
    - Génération d'un script de configuration des makefiles
  - Autoheader
    - Génération de fichiers d'entêtes
  - Libtool
    - Génération de librairies portables
  - Aclocal
  - Autoscan
  - Autoreconf
  - …

Schéma global



Exemple simple

- Le programme qui fait coucou

src/main.c

```
#include <stdio.h>
void main(void){
        printf("Coucou\n");
}
```

configure.ac

- Un modèle de fichier (configure.scan) peut-être obtenu par l'exécution de la commande « autoscan »

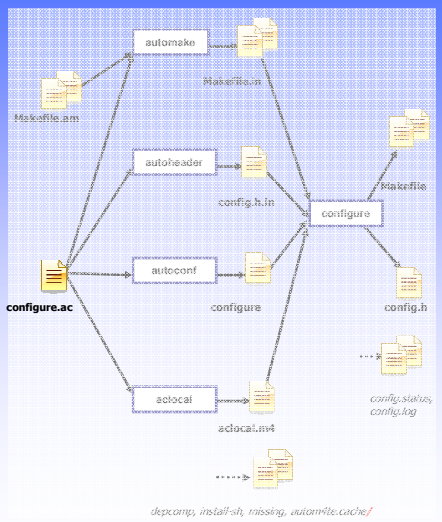
```
AC_PREREQ(2.59)
AC_INIT(coucou, 0.1, muller@lsiit.u-strasbg.fr)
AM_INIT_AUTOMAKE

# Checks for programs.
AC_PROG_CC
AC_PROG_MAKE_SET

# Checks for libraries.
# AC_SEARCH_LIBS(fonction, nom-librairie)

# Checks for header files.
AC_CONFIG_HEADERS([config.h])

AC_CONFIG_FILES([Makefile src/Makefile])
AC_OUTPUT
```



configure.ac

```
AC_PREREQ(2.61)

AC_INIT(coucou, 0.1, muller@lsiit.u-strasbg.fr)

AM_INIT_AUTOMAKE

# Checks for programs.
AC_PROG_CC

AC_PROG_MAKE_SET

# Checks for libraries.
# AC_SEARCH_LIBS(fonction, nom-librairie)

# Checks for header files.
AC_CONFIG_HEADERS([config.h])
.
AC_CONFIG_FILES([Makefile src/Makefile])

AC_OUTPUT
```

- Version d'autoconf ayant été utilisée
- Initialisation d'autoconf. Spécifie le nom du paquet, la version, le nom du développeur
- Initialisation d'automake (sans options ici)
- Teste la présence du compilateur C
- Teste la présence du programme make
- Teste la présence d'une fonction dans une librairie
- Spécifie le nom du fichier d'entêtes à générer
- Spécifie le nom des fichiers Makefiles à générer
- Initialise la génération des fichiers spécifiés

Makefile.am

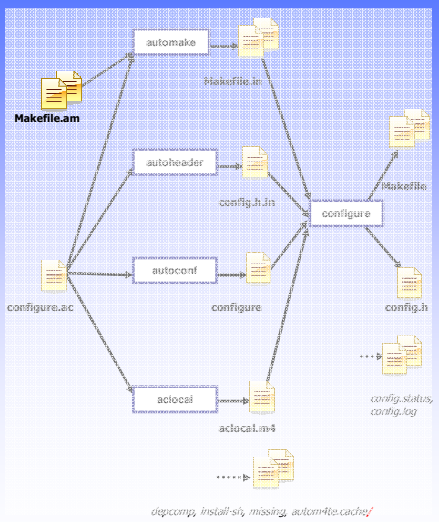
- Makefile.am

  SUBDIRS = src

- src/Makefile.am

  bin_PROGRAMS = coucou

  coucou_SOURCES = main.c



Makefile.am

- Makefile.am

  SUBDIRS = src

- src/Makefile.am

  bin_PROGRAMS = coucou

  coucou_SOURCES = main.c

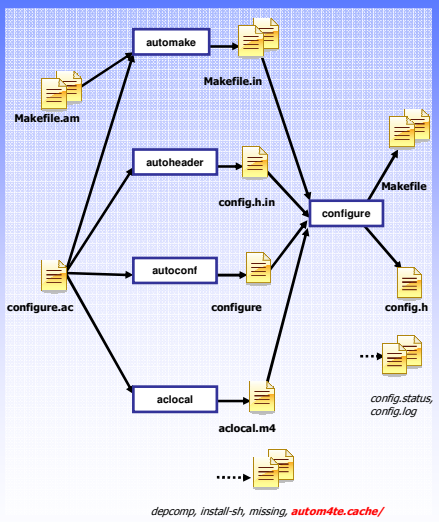
- Sous-répertoires à prendre en compte pour la génération de makefiles

- Nom des exécutables à générer
  - s'il y a plusieurs exécutables les séparer par un espace

- Fichiers à prendre en compte pour générer l'exécutable « coucou »
  - s'il y a plusieurs fichiers les séparer par un espace

Mode opératoire

- aclocal
  - Recherche d'éventuelles macros m4 dans configure.ac
  - Générer le fichier aclocal.m4
- autoheader
  - Génère config.h.in à partir de configure.ac
  - Permet la définition de variables propres au paquet
- autoconf
  - Génère le script configure à partir de configure.ac
- touch AUTHORS ChangeLog NEWS README
- automake --add-missing --copy
- ./configure
- make distcheck



Mode opératoire

- aclocal
  - Recherche d'éventuelles macros m4 dans configure.ac
  - Générer le fichier aclocal.m4
- autoheader
  - Génère config.h.in à partir de configure.ac
  - Permet la définition de variables propres au paquet
- autoconf
  - Génère le script configure à partir de configure.ac
- touch AUTHORS ChangeLog NEWS README
- automake --add-missing --copy
- ./configure
- make distcheck

- Les cinq premières étapes peuvent être remplacées par autoreconf --install
- Sans ces fichiers le paquet ne sera pas au format GNU et automake refuse de continuer
- Automake essaye de copier les fichiers manquants (INSTALL, COPYING)
- Génération des Makefiles
- Génération d'un paquet (archive .tgz dans le répertoire racine du projet)

Génération de librairies : exemple

- Le programme qui fait coucou en plus élaboré
- src/main.c

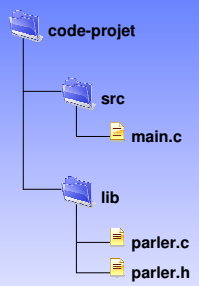
```
#include <stdio.h>
#include "parler.h"
void main(void){
    dire_coucou();
}
```

- lib/parler.c

```
#include <stdio.h>
#include "parler.h"
void dire_coucou (void){
    printf("Coucou\n");
}
```

- lib/parler.h

```
#ifndef PARLER_H
#define PARLER_H
void dire_coucou(void);
#endif
```



Génération de librairies statiques

- Dans configure.ac

```
AC_PROG_RANLIB
AC_CONFIG_FILES([Makefile
                 src/Makefile lib/Makefile])
```

- Dans src/Makefile.am

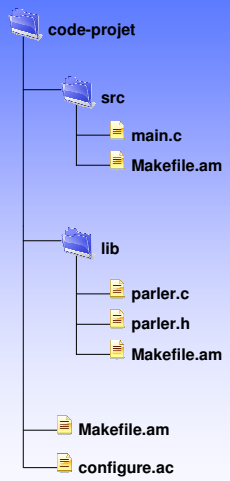
```
coucou_LDADD =
    ${top_builddir}/lib/libparler.a
coucou_LDFLAGS = -L${top_builddir}/lib
coucou_CPPFLAGS = -I${top_srcdir}/lib
bin_PROGRAMS = coucou
coucou_SOURCES= main.c parler.h
```

- Dans lib/Makefile.am

```
noinst_LIBRARIES = libparler.a
libparler_a_SOURCES = parler.c
    parler.h
```

- Dans Makefile.am

```
SUBDIRS = lib src
```



Génération de librairies dynamiques - Libtool

- Format des librairies dynamiques propres à chaque système
  - libpipo.so
  - libpipo.dll
  - libpipo.dylib
- Construction des librairies nécessitent des flags différents selon l'OS
  - -shared, -fPIC
  - -G, -KPIC
- Format Libtool est une abstraction de tous les formats
  - libpipo.la (libtool archive)
  - Lors de la compilation/édition de lien, on a conversion vers le format approprié

Génération de librairies dynamiques - Libtool

- Dans configure.ac

```
AC_PROG_LIBTOOL
AC_CONFIG_FILES([Makefile src/Makefile lib/Makefile])
```

- Dans src/Makefile.am

```
bin_PROGRAMS= coucou
coucou_SOURCES= main.c parler.h
coucou_LDADD = ${top_builddir}/lib/libparler.la
coucou_LDFLAGS = -L${top_builddir}/lib
coucou_CPPFLAGS = -I${top_srcdir}/lib
```

- Dans lib/Makefile.am

```
lib_LTLIBRARIES = libparler.la
libparler_la_SOURCES = parler.c parler.h
Include_HEADERS = parler.h
```

- Dans Makefile.am

```
SUBDIRS = lib src
```

- Au niveau des commandes rajouter « libtoolize -f »

Les commandes de l'utilisateur

- ./configure
- make all
- make install
- make install-strip
- make uninstall
- make clean
- make distclean
- make check
- make installcheck
- make dist
- make distcheck

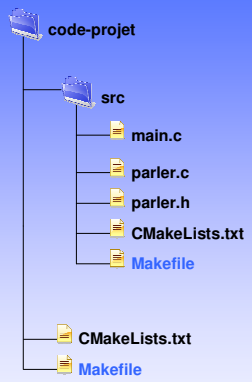
CMake

Généralités

- Plus simple que les Autotools
  - Écrit en C++
  - Une seule syntaxe
  - Un seul type de fichier
    - CMakeLists.txt
  - Une seule commande
    - cmake
- Plus compréhensible lors de l'exécution
- Plus portable que les autotools
  - Makefiles, projets KDevelop, Microsoft Visual Studio, Borland
- Pérennité
  - Adopté par le projet KDE depuis la version 4
  - Développé depuis 2001 par Kitware

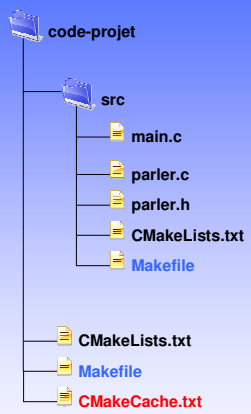
Mode opératoire

- Créer et paramétrer un fichier CMakeLists.txt dans chaque répertoire du projet
- Utiliser la commande cmake
  - Construction dans les sources du projet
    - cmake .
  - Construction hors des sources du projet
    - cmake /chemin_code_projet
- Génération des fichiers Makefile, ou .dsp, ou .sln, ...
  - Utilisables par les outils natifs



Fichier cache - CMakeCache.txt

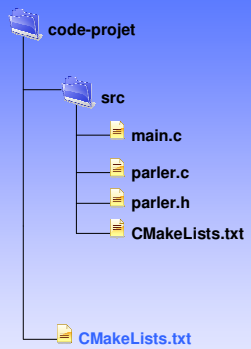
- Généré lors du premier appel à cmake
  - Configuration de la construction
  - variables et leurs valeurs
  - En cas de problème
    - Modifier manuellement
    - Éditer ce fichier grâce à ccmake



CMakeLists.txt

```
PROJECT(coucou)
SUBDIRS(src)
```

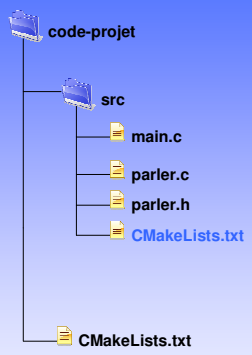
- Indique le nom du projet
  - Sera utilisé lors de la génération du paquet
- Indique les répertoires à prendre en compte
  - Traite les répertoires dans l'ordre indiqué



CMakeLists.txt

```
ADD_EXECUTABLE(coucou main.c parler.c)
TARGET_LINK_LIBRARIES(produit m)
```

- Indique le nom de l'exécutable et les fichiers sources servant à le générer
- Indique les librairies à lier



Librairies

- Création d'une librairie statique

  ADD_LIBRARY(Libparler STATIC parler.c)

- Création d'une librairie dynamique

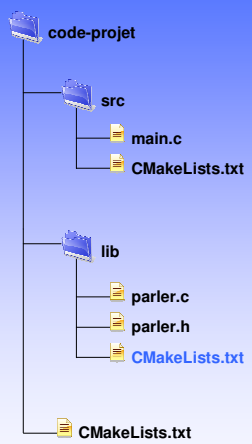
  ADD_LIBRARY(Libparler SHARED parler.c)

- Pour que main.c puisse trouver le fichier d'entêtes

  INCLUDE_DIRECTORIES(${CMAKE_SOURCE_DIR}/lib)

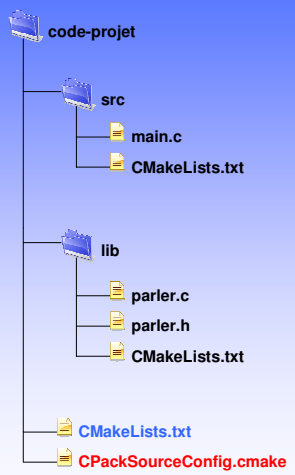
- Indique les répertoires à prendre en compte
  - Traite les répertoires dans l'ordre indiqué

  SUBDIRS(lib src)



Génération d'un paquet

- Appel à l'outil CPack
  INCLUDE(CPack)
- cmake puis :
  - make package ou cpack
    - Paquet comprenant les binaires
  - make package_source
    - Paquet comprenant les sources
- Fichier CPackSourceConfig.cmake
  - Paramètres système relevés et utilisés par cpack



Macros

- Composition de commandes existantes

```
MACRO (CREATE_EXECUTABLE NAME SOURCES LIBRARIES)
      ADD_EXECUTABLE(${NAME} ${SOURCES})
      TARGET_LINK_LIBRARIES(${NAME} ${LIBRARIES})
ENDMACRO(CREATE_EXECUTABLE)

ADD_LIBRARY(LibParler parler.c)
CREATE_EXECUTABLE(coucou main.c LibParler)
```

Création de variables booléennes - Utilisation en ligne de commande

- Définir des variables pour des tests conditionnels

```
OPTION(DEBUG "programme avec débogage"   OFF)
IF(DEBUG)
        SET_SOURCE_FILES_PROPERTIES(parler.c main.c COMPILE_FLAGS –DDEBUG)
ENDIF(DEBUG)
```

➢ Activation de l'affichage des message de débogage
  - cmake –DDEBUG:BOOL=ON

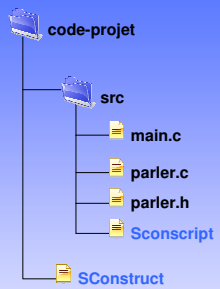
Scons

Généralités

- Plus simple que les Autotools
  - Lancé en 2000 par le projet Blender
  - Une seule syntaxe
  - Un seul type de fichier
    - SConstruct/SConscript
  - Une seule commande
    - scons
- Portable
  - Multi-plateforme (Linux, windows, MacOS)
- Facilement extensible et puissant
  - Écrit en Python
- Bien documenté

Mode opératoire

- Créer et paramétrer un fichier SConstruct à la racine du projet
- Créer et paramétrer un fichier SConscript dans chaque sous-répertoire du projet
- Utiliser la commande scons
- Compilation en utilisant les outils de construction du système cible
  - Makefile sur Linux
  - CI sur Windows



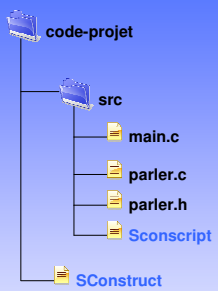
Les fichiers Sconscript

- Sconstruct

  Sconscript (‘src/Sconscript’)

- Src/Sconscript

  Program(‘coucou’, [‘main.c’,’parler.c’])



Librairies

- Création d’une librairie statique

  StaticLibrary(‘parler’,’parler.c’)

- Création d’une librairie dynamique

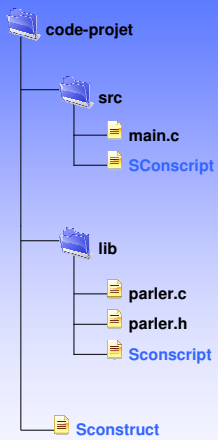
  SharedLibrary(‘parler’,’parler.c’)

- Trouve le fichier d’entêtes et la librairie

  Program('main.c', LIBS='parler', LIBPATH='../lib', CPPPATH ='../lib')

- Indique les répertoires à prendre en compte
  - Ne tient pas compte de l’ordre indiqué

  Sconscript ([‘src/Sconscript’,’ lib/Sconscript’])



Environnement

- Possibilité de définir des environnements de construction
  - Ensemble de variables
- Personnalisation du traitement de chaque fichier

```
opt = Environment(CCFLAGS = '-O2')
dbg = Environment(CCFLAGS = '-g')

opt.Program('foo', 'foo.c')
dbg.Program('bar', 'bar.c')
```

- A l'exécution

```
% scons -Q
cc -o bar.o -c -g bar.c
cc -o bar bar.o
cc -o foo.o -c -O2 foo.c
cc -o foo foo.o
```


Environnement

- Afficher les variables d'un environnement

```
env = Environment()
dict = env.Dictionary()
keys = dict.keys()
keys.sort()
for key in keys:
    print "construction variable = '%s', value = '%s'" % (key, dict[key])
```

- Faire une copie d'un environnement

```
env = Environment(CC = 'gcc')
opt = env.Clone(CCFLAGS = '-O2')
dbg = env.Clone(CCFLAGS = '-g')
env.Program('foo', 'foo.c')
opt.Program('foo-opt', 'foo.c')
dbg.Program('foo-dbg', 'foo.c')
```

Reconstruction sélective

- Plusieurs possibilités de choix de reconstruction
  - Timestamp comme Makefiles
    - Decider('timestamp-newer')
    - Decider(‘make')
    - Decider('timestamp-match')
  - Checksum md5
    - Méthode par défaut
    - Decider('MD5')
  - Combinaison des deux
    - Decider('MD5-timestamp')

Divers

- Construction hors des sources

```
SConscript('src/SConscript', variant_dir='build')
```

- Nettoyage des sources après construction

```
scons –c
```

- Manipulation des listes

```
Program('program', Glob('*.c'))
Program('program', Split('main.c file1.c file2.c'))
```

- Déterminer la localisation d’une librairie

```
env = Environment()
env['CPPPATH'] = ['/lib/compat']
env.ParseConfig("pkg-config x11 --cflags --libs")
```

  - Résultat placé dans dans CPPPATH

Conclusion

Conclusion

- Makefiles
  - Facile d'utilisation
  - Limité au monde linux/unix
  - Pas portable
- Autotools
  - Portabilité
  - Compliqué à mettre en œuvre
  - Limité pour Windows
- Cmake
  - Simple d'utilisation
  - Soutenu par quelques gros projets
  - Portabilité correcte
- Scons
  - Simple d'utilisation
  - Scripts python
  - Bonne portabilité
  - Bien documenté

Références

- Makefiles
  - Partout sur Internet
- Autotools
  - Documentation officielle
    - www.gnu.org/software/automake/
    - http://www.gnu.org/software/autoconf/
    - http://www.gnu.org/software/libtool/
  - Vulgarisation
    - Articles d'Yves Mettier - Linux Magazine 75, 76, 77, 79, 80, 81, 84
    - Présentation d'Alexandre Duret-Lutz
- CMake
  - Documentation officielle
    - http://www.cmake.org/
  - Vulgarisation
    - Article d'Alexandre Courbot – Linux Magazine 92
    - Articles de Patrice Leygnac – Linux Magazine 105, 108
- Scons
  - Documentation officielle
    - http://www.scons.org/