M2 I2L - 2009/2010

Environnements de développement libres

Makefile, Cmake, Ctest et Swig

David DUVIVIER
duvivier@lil.univ-littoral.fr
D'après le cours de ...
Eric RAMAT
ramat@lil.univ-littoral.fr

Partie 1 : Makefile

Introduction

- Les Makefiles sont des fichiers utilisés par le programme make pour exécuter un ensemble d'actions
- Plusieurs utilitaires :
  - GNU make
  - mais aussi : gmake, nmake, tmake, ...
- Make est un outil qui contrôle la génération d'exécutables et les autres fichiers non-source d'un programme à partir des fichiers sources.

Introduction

- Gérer les dépendances entre fichiers à : construire / compiler / “linker” / exécuter
- Automatiser la construction / l'exécution en : reconstruisant ce qui est nécessaire et uniquement suffisant
- Rapidité de construction par rapport au processus “à la main”
- Exemples :
    - .o/.c/.h : bibliothèque, archive, installation
    - .tex/.dvi/.ps : génération de documentation

Principes

- Un graphe de dépendances va être construit à partir :
  - de cibles (fichier ou simple nom)
  - de leurs dépendances
  - des tâches à réaliser
- Qu'est qu'une dépendance ?
  - un élément qui nécessite l'exécution des tâches s'il n'est pas à jour par rapport à la cible
- Comment détermine-t-on qu'on est à jour ?
  - par les dates de dernière modification

Principes

- L'ensemble des informations est centralisé dans un fichier, interprété par la commande make :
  - Nom par défaut : makefile ou Makefile
  - Constitué de règles
- Une règle a la forme :
  - cible : liste des dépendances
  - <tab> tâches (en sh)
- Des raccourcis et des variables d'environnement permettent de décrire plus facilement les règles, les cibles, les dépendances et les tâches

Premier exemple

```
HELLOWORLD: HELLOWORLD.O
<TAB> GCC @ HELLOWORLD HELLOWORLD.O

HELLOWORLD.O : HELLOWORLD.C HELLOWORLD.H
<TAB> GCC -C HELLOWORLD.C
```

Syntaxe

- Schéma général :

```
CIBLE1 ... CIBLEN : PREREQUIS1 ... PREREQUISP
<TAB>                    COMMANDE1 ; ... ; COMMANDEC
```

- ou bien

```
CIBLE1 ... CIBLEN : PREREQUIS1 ... PREREQUISP ;
COMMANDE1 ; COMMANDE2 ; ... ; COMMANDEC
```

- ou encore

```
CIBLE1 ... CIBLEN : PREREQUIS1 ... PREREQUISP
 <TAB>                  COMMANDE1
 <TAB>                  ...
 <TAB>                  COMMANDEC
```

- Tous les cas intermédiaires sont possibles ...
- Les cibles doivent être séparées par des espaces ou tabulations ainsi que les dépendances.

Interprétation

- Pour décider d'exécuter les tâches d'une règle, il faut vérifier au préalable si une des dépendances nécessite une mise en cohérence par rapport à la cible.
- Les commandes qui composent la tâches sont alors exécutées séquentiellement
- Si une commande provoque à une erreur, alors la cible n'est pas rendue cohérente et TOUT le processus s'arrête.
- Une cible est **incohérente** lorsque :
  - un de ses dépendants est incohérent,
  - la cible n'existe pas,
  - sa date de dernière modification est antérieure à celles de ses dépendances

La commande Make

- make nomDeCible
- make -f monMakefile
- make -i
  - ignorer globalement toutes les erreurs qui peuvent survenir
- make -n
  - montrer ce que l'on ferait, mais ne pas le faire...
- make
  - Makefile ou makefile dans le répertoire courant
  - reconstruction de la première cible trouvée

Autres éléments de syntaxe

- Passage à la ligne
  - Les cibles et les prérequis doivent se trouver sur une même ligne.
  - Sauf en utilisant le caractère “ \ “
    - Lorsque make lit un fichier makefile, il ignore toutes les occurrences du caractère “ \ “ suivi d'un caractère “ \n “ (newline).
  - Donc la règle :

```
CIBLE1 CIBLE2 \
CIBLE3 ... CIBLEN : \
PREREQUIS1\
PREREQUIS2 PREREQUIS3 \
... PREREQUISP ; COMMANDE1 ; COMMANDE2 COMMANDE3 ; ... ; COMMANDEB ; \
COMMANDEC
```

Autres éléments de syntaxe

- Commentaire
  - Toute suite de caractères après un “ # “.
- Le caractère TAB
  - Une ligne commence par TAB, ce qui suit est interprété comme une commande shell.
  - Une série d'espace n'a pas le même effet
  - Un TAB “ esthétique “ en début de ligne peut jouer des tours...
- Commande invisible
  - Le caractère @ placé devant une commande empêche l'affichage de l'intitulé de la commande à la console.

Variables

- Il y a plusieurs types de variables pour make :
  - les variables passées par commande
  - les variables d'environnement
  - les variables définies dans le makefile
  - les variables automatiques

Variables passées par commande

- make TOTO=objet.c
- dans le fichier makefile, toutes les occurrences de $(TOTO) ou de ${TOTO} seront remplacées par objet.c.

Variables d'environnement

- Toutes les variables du shell lançant make sont accessibles
  - echo $(VARIABLE) renvoie la valeur de la variable d'environnement VARIABLE.
- Si la commande make est lancée par un shell donné (bash, zsh, sh...), toutes les variables d'environnement de ce shell peuvent être utilisées dans le makefile.
- La seule exception est la variable SHELL qui, par défaut, vaut toujours /bin/sh, mais qui peut être modifiée dans le makefile.

Variables définies dans le Makefile

- Définition
  - var=valeur
- Mécanisme : la substitution de la variable var
  - toutes les occurrences de $(var) ou de $var seront remplacées par valeur
- Attention, la substitution est récursive !

Manipulation des variables

- $variable correspond à une variable du makefile...
- le $$ peut être utilisé pour passer une expression de variable commençant par $ au shell.

```
AFFICHE:
        FOR FIC IN *.C; DO LS $$FIC; DONE

MATRIX% MAKE AFFICHE
FOR FIC IN *.C; DO LS $FIC; DONE
CONVERT2SENS.C CONVERTIR.C OUTILS.C
```

Manipulation de variables

- Pour pouvoir ajouter quelque chose à une variable (à substitution récursive ou simple), on utilise l'opérateur binaire « += » :
  - variable = valeur
  - variable += davantage
- est équivalent à :
  - temp = valeur
  - variable = $(temp) + davantage
- sauf qu'aucune variable temp n'est définie

Exemple

```
CC = gcc
CFLAGS = -c
OBJS = convertir.o outils.o convert2sens.o

convertir: $(OBJS)
        $(CC) $(OBJS) -o convertir
convert2sens.o: convert2sens.c convertir.h
outils.h
        $(CC) $(CFLAGS) convert2sens.c
...
```

Jokers et règles à motifs

- * et ?, [a-z] et autres peuvent être utilisés dans :
  - les commandes shell (comme d'habitude)
  - les cibles
  - les prérequis
- On peut ainsi créer des règles à motifs :

```
*.o : *.c
	gcc -c *.c
```

Variables automatiques

- Ces variables sont définies et remises à jour par make.
- Elles ne peuvent être utilisées que dans une règle et désignent une des cibles, les prérequis, un sous-ensemble de prérequis, ...
- $@ : remplacé par le nom du fichier cible.
  - s'il y a plusieurs noms dans la cible, alors $@ est remplacé par celui des noms dans la cible qui a déclenché l'application de la règle.

```
CONVERTIR: $(OBJS)
                $(CC) $(OBJS) -o $@
```

Variables automatiques

- $* : préfixe de la cible
- $^ : remplacé par la liste des prérequis, chaque nom étant séparé par une espace :
  - si un certain prérequis a été répété plusieurs fois, alors il n'apparaît qu'une seule fois dans $^.
  - si on veut que l'ordre des éléments et le nombre des répétitions soient préservés, il faudra utiliser $+ au lieu de $^.

Variables automatiques

- $< désigne le premier prérequis.
  - nom du fichier qui a causé l'action associée à la cible (celui qui serait utilisé par la règle implicite)

```
CONVERT2SENS.O : CONVERT2SENS.C
	$(CC) -c $<
```

Variables automatiques

$? est remplacé par la liste des prérequis qui sont plus jeunes qu'au moins une des cibles.

- en fait $? désigne la liste des prérequis qui sont responsables de l'application de la commande de la règle.

```
PRINTC : *.C
        @ECHO $? : FICHIERS C


MATRIX% MAKE PRINTC
CONVERT2SENS.C CONVERTIR.C OUTILS.C : FICHIERS C
```

- Il existe d'autres variables automatiques ($%, $+, ...)

Makefile récursif

- Considérons des sous-répertoires où chacun contient un makefile
- Pour utiliser les makefiles dans chaque répertoire :
  - on peut aller dans chacun de ces répertoires (cd)
  - taper les commandes make adéquats
- Lors de l'appel récursif de make, la commande « make » utilisée est celle définie par défaut sur le système :

```
${MAKE} -C SUBREP ARG_1 ARG_2 ARG_3 ... ARG_N
```

Règles implicites

```
CONVERT2SENS.O : CONVERT2SENS.C
                $(CC) -C $<
```

- est une règle intéressante, mais on pourrait l'appliquer à tout fichier .c pour le lier au .o correspondant
- il existe en fait une telle règle, dite implicite :

```
    .C.O:
            $(CC) -C $<
OU
    %.O:%.C
            $(CC) -C $<
```

Exemple

```
CC=gcc
CFLAGS=-Wall -ansi
LDFLAGS=-Wall -ansi
EXEC=hello
all: $(EXEC)
hello: hello.o main.o
	$(CC) -o $@ $^ $(LDFLAGS)
main.o: hello.h
%.o: %.c
	$(CC) -o $@ -c $< $(CFLAGS)
clean:
	rm -rf *.o
```

Partie 2 : CMake

Qu'est ce que CMake ?

- Outil d’aide au processus de compilation (build),multi plates-formes et open source
- Développé depuis 2001 par Kitware (VTK – Visualization ToolKit)
- Adopté par les développeurs de KDE
- Capable de générer des makefile/unix, des projets MS Visual Studio, Borland … qui seront utilisés par les outils natifs lors de la compilation
- Nombreuses commandes permettant de localiser les fichiers d'entêtes, les bibliothèques, les exécutables

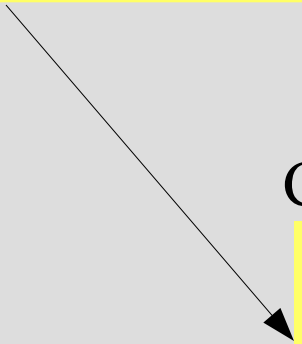
Qu'est ce que CMake ?

- Propose de nombreuses extensions pour localiser X, Qt, OpenGL, Swig, …
- Propose des interfaces de test (CTest) et de packaging (Cpack)
- Site Web : http://www.cmake.org
- Paquet Debian : version 2.6-patch 0
- Installation : apt-get install cmake

Un simple exemple

Makefile

```
main.o: main.c main.h
        cc -c main.c
MyProgram: main.o
        cc -o MyProgram main.o -lm -lz
```



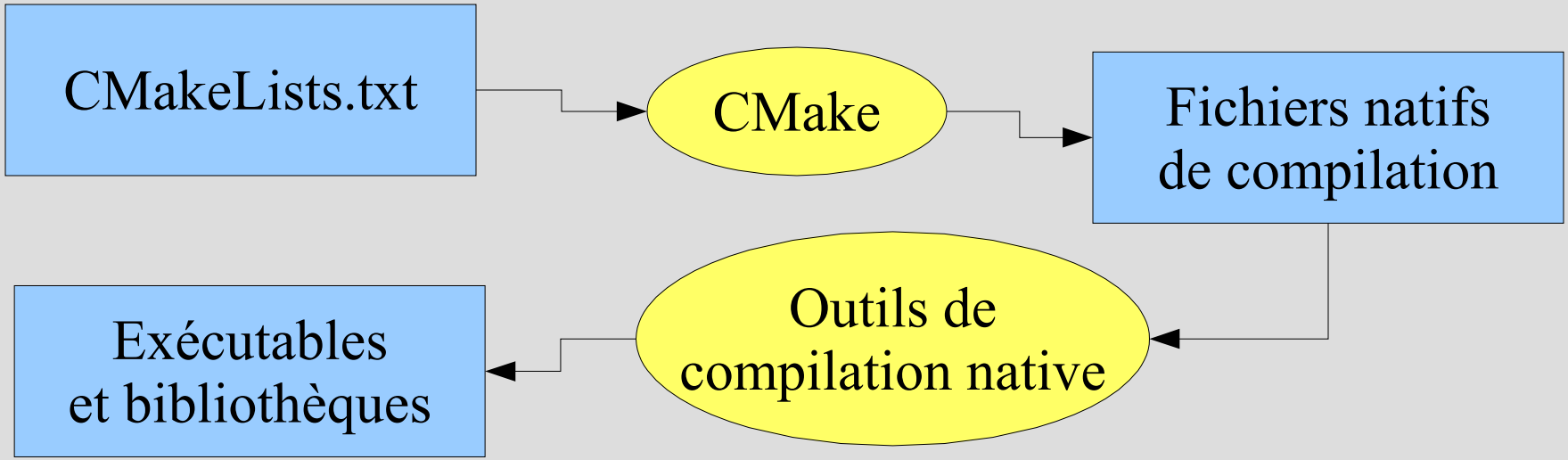
CMakeLists.txt

```
PROJECT(MyProject C)
ADD_EXECUTABLE(MyProgram main.c)
TARGET_LINK_LIBRARIES(MyProgram z m)
```

Le fichier “Makefile” est remplacé par “CMakeLists.txt”

Schéma de compilation

- Capable de gérer le processus de compilation indépendamment du SE et du compilateur
- Connaissance de très nombreuses plates-formes et outils
- L'utilisateur configure son *build* avec CMake



Structure - répertoire

- L'arborescence des sources contient :
  - fichier d'entrée de CMake (CMakeLists.txt)
  - fichiers sources du programme (*.cxx, par exemple)
- L'arborescence des produits de la compilation contient :
  - fichiers natifs de compilation (Makefile, *.dsp pour Visual Studio)
  - les bibliothèques et les exécutables (*.exe, par exemple)
- Les deux arborescences peuvent être :
  - dans le même répertoire (*in-source* build)
  - dans des répertoires différents (*out-of-source* build) :
    - src : les sources (cpp, hpp, java, py, ...) et les CMakeLists.txt
    - build : les produits de la compilation (.so, .a, exe, ...)

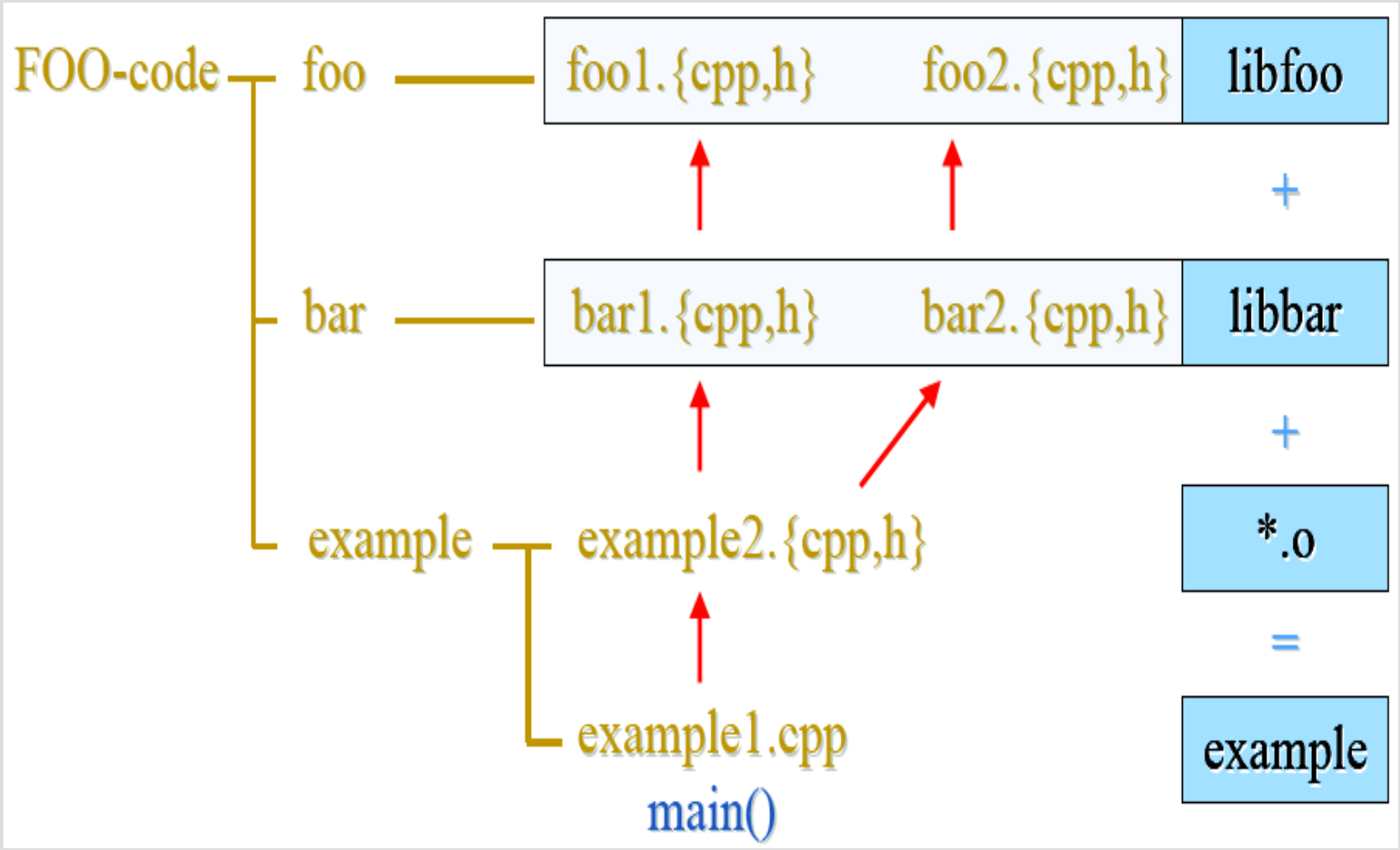
Eléments de syntaxe

- langage de type script :
  - Commentaire : # commentaire jusqu'à la fin de la ligne
  - Commande : COMMAND (arg1 arg2 ...)
  - Liste : A;B;C
  - Variable : ${VAR}
  - Structure de contrôle : IF (CONDITION) FOREACH(v A B C)
- Détection de bibliothèques : FIND_LIBRARY(...)

Le cache de CMake

- Résume l'ensemble de la configuration du build
- Fichier CMakeCache.txt situé à la racine du projet
- Créé et mis à jour lors des phases de configuration réalisées par CMake
- Les variables incontournables :
  - CMAKE_BUILD_TYPE=[Debug, Release]
  - CMAKE_INSTALL_PREFIX=[/usr/local, C:\Program Files\ViSP]
  - CMAKE_VERBOSE_MAKEFILE=[OFF, ON]

Un exemple


FOO-code
foo
foo1.{cpp,h}
foo2.{cpp,h}
libfoo
+
bar
bar1.{cpp,h}
bar2.{cpp,h}
libbar
+
example
example2.{cpp,h}
*.o
=
example1.cpp
main()
example

Un exemple

```
PROJECT(projectname [CXX] [C] [JAVA])
```

FOO-code

CMakeLists.txt

```
PROJECT(FOO)
SUBDIRS(foo bar example)
INCLUDE_DIRECTORIES(${CMAKE_SOURCE_DIR}/foo
  ${CMAKE_SOURCE_DIR}/bar)
SET(LIBRARY_OUTPUT_PATH ${FOO_BINARY_DIR}/lib)
```

foo - CMakeLists.txt

```
ADD_LIBRARY(foo foo1.cpp foo2.cpp)
```

bar - CMakeLists.txt

```
ADD_LIBRARY(bar bar1.cpp bar2.cpp)
TARGET_LINK_LIBRARIES(bar foo)
```

example - CMakeLists.txt

```
ADD_EXECUTABLE(example example1.cpp example2.cpp)
TARGET_LINK_LIBRARIES(example bar)
```

Un exemple

- Attention, les sources des sous-répertoires ne sont incluses dans le projet que si un CMakeLists.txt est présent dans le répertoire FOO-code :
  - ADD_SUBDIRECTORY(foo)
  - ADD_SUBDIRECTORY(bar)
  - ADD_SUBDIRECTORY(example)
  - ou SUBDIRS(foo bar example)

Installation

- Par défaut, les éléments (.h, .so, les exécutables, ...) sont installés dans /usr/local/
- Dans le cas de la séparation du build des sources (construction dans le répertoire build) :
  - cmake -DCMAKE_INSTALL_PREFIX=$HOME/work/usr/ ..
  - la variable ${CMAKE_INSTALL_PREFIX} est modifiée
- La commande d'installation des résultats de la compilation :
  - INSTALL(TARGETS example DESTINATION ${CMAKE_INSTALL_PREFIX}/bin)

Installation

- Installation des headers :
  - SET(BAR_H bar1.h bar2.h)
  - INSTALL(FILES ${BAR_H} DESTINATION ${CMAKE_INSTALL_PREFIX}/include
- La même technique peut être utilisée pour n'importe quel fichier

PkgConfig

- A l'installation de bibliothèques, on peut avoir à notre disposition des .pc : informations sur l'emplacement des fichiers (.h, .a, .so, ...)
- Exemple :
  - pkg-config --libs glibmm-2.4
    - -lglibmm-2.4 -lgobject-2.0 -lsigc-2.0 -lglib-2.0
  - pkg-config --cflags glibmm-2.4
    - -I/usr/include/glibmm-2.4 -I/usr/lib/glibmm-2.4/include -I/usr/include/sigc++-2.0 -I/usr/lib/sigc++-2.0/include -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include

PkgConfig

- Recherche des paquets :
  FIND_PACKAGE(PkgConfig)
  PKG_CHECK_MODULES(GLIBMM glibmm-2.4)
- Attention, ces commandes nécessitent un fichier .cmake avec des macros installé dans un répertoire cmake à la racine de votre projet (*/usr/share/cmake-2.6/Modules/FindPkgConfig.cmake*)
- Utilisation des informations (--libs et –cflags) :
  - INCLUDE_DIRECTORIES(${GLIBMM_INCLUDE_DIRS})
  - LINK_DIRECTORIES(${GLIBMM_LIBRARY_DIRS})
  - ADD_EXECUTABLE(example example1.cpp example2.cpp)
  - TARGET_LINK_LIBRARIES(example ${GLIBMM_LIBRARIES})

PkgConfig

- Les sources example1.cpp et example2.cpp peuvent maintenant inclure des headers de la glibmm
- Les variables créées par la commande PKG_CHECK_MODULES sont disponibles dans les sous-répertoires

Partie 3 : CTest et Boost

Qu'est ce que CTest ?

- Fait partie de la distribution CMake
- Peut-être utilisé avec ou sans CMake
- Permet de réaliser plusieurs types de test sur le code source :
  - Exécution de tests prédéfinis,
  - Exécution de tests avancés:
    - Couverture de code : uniquement avec g++ -fprofile-arcs -ftest-coverage
    - Etat de la mémoire : utilisation de purify ou valgrind

CTest

- Les tests doivent être activés dans le CMakeLists.txt principal :
  - ENABLE_TESTING()
- Le(s) programme(s) de test est/sont un/des exécutable(s) comme les autres
  - ADD_EXECUTABLE(test1 test1.cpp)
  - TARGET_LINK_LIBRARIES(test1 ${Boost_UNITTESTFRAMEWORK})
  - ADD_TEST(apptest1 test1)
- Ici, on linke l'exécutable avec une bibliothèque de tests unitaires : Boost::UnitTestFramework

CTest

- Liste des tests :
  - **ctest -N**
- Exécution des tests :
  - **make test** ou **ctest**
  - tous tests sont exécutés et un rapport est généré
- Exécution d'un sous-ensemble des tests :
  - ctest -I 2,3
  - les tests 2 et 3 sont exécutés
- Un répertoire est crée dans le build avec les rapports : Testing/Temporary
  - LastTest.log et LastTestsFailed.log

Boost::Test

- voir présentation de Boost

Exemple Moyenne

```
PROJECT(MOYENNE CXX C)
ENABLE_TESTING()

FIND_PACKAGE(Boost)
IF (Boost_FOUND)
  FIND_LIBRARY(Boost_UNITTESTFRAMEWORK NAMES
     boost_unit_test_framework
    boost_unit_test_framework-mt PATHS Boost_LIBRARY_DIRS)
  IF (Boost_UNITTESTFRAMEWORK)
    SET(HAVE_UNITTESTFRAMEWORK 1 CACHE INTERNAL "" FORCE)
  ELSE (Boost_UNITTESTFRAMEWORK)
    SET(HAVE_UNITTESTFRAMEWORK 0 CACHE INTERNAL "" FORCE)
  ENDIF (Boost_UNITTESTFRAMEWORK)
ENDIF (Boost_FOUND)

ADD_SUBDIRECTORY(src)
```

CMakeLists.txt - racine

Exemple Moyenne

```
INCLUDE_DIRECTORIES(${Boost_INCLUDE_DIRS})

LINK_DIRECTORIES(${Boost_LIBRARY_DIRS})

ADD_EXECUTABLE(test1 test.cpp moyenne.cpp)
ADD_TEST(moyenne_test test1)
TARGET_LINK_LIBRARIES(test1 ${Boost_UNITTESTFRAMEWORK})
```

CMakeLists.txt - src

Exemple Moyenne

- Il faut développer un main dans le code à tester (même s'il est vide)
- Créer un répertoire build, taper : cmake ..
- Lancer : make
- Puis, make test

Partie 4 : Swig

Swig, c'est quoi ?

- Outil de création de wrapper de code C/C++ pour des langages à base de machines virtuelles ou de scripts :
  - Python, Java, C#, Ruby
  - Perl, PHP, ...
- Définition de fichiers d'interface (comme IDL pour Corba)

Fichier d'interface
un exemple simple

- Exemple d'une simple fonction C :
  int f(int p);
- fichier interface : exemple.i
  - définition du nom de module :
    %module exemple
  - inclusion du lien avec le fichier C :
    %{
    #include “toto.h”
    %}
  - entête de la fonction à wrapper :
    int f(int p);

Comment lancer Swig ?

- swig [option] nom_de_fichier
- nom_de_fichier : fichier d'interface (.i)
- options :
  - -c++ pour activer le mode C++ sinon C
  - -<langage> précise le langage cible
- les fichiers générés :
  - le fichier de wrapping en C/C++ : <fichier.i>_wrap.cxx (en mode C++) ou .c (en mode C)
  - le fichier de wrapping dans le langage cible : <fichier.i>.<extension_langage>

Fichier d'interface
un exemple simple

- Création des fichiers de wrapping pour Python :
  swig -c++ -python exemple.i
- Compilation du wrapper sous forme d'une librairie dynamique :
  - g++ -Wall -g -I/usr/include/python2.4 -c exemple_wrap.cxx -o exemple_wrap.o
  - g++ -shared -fpic -o _exemple.so exemple_wrap.o
- Attention pour Python, le nom de la librairie est préfixé par un *underscore*

Fichier d'interface
un exemple simple

- L'utilisation de la librairie dans Python consiste à faire un import <nom_librairie> sans l'*underscore* devant

```
import exemple
exemple.f(5)
```

Swig et les classes

- Deux solutions :
  - définition complète ou partielle de la classe à wrapper
  - utilisation de proxies générés automatiquement

Director

- Le mécanisme de Director génére des wrappers de type proxy :
  - tout appel passe par un proxy qui fait le lien avec l'objet C++
- Ce mécanisme peut être appliqué :
  - à un module :

    %module(directors=1) exemple
  - ou seulement à une classe :

    %feature(“director”) classe;

Director - suite

- Avec Director, il faut ajouter une commande %include pour chaque fichier d'entête définissant une classe :
  %include “classe.hpp”
- Cette commande vient compléter l'autre include qui sert uniquement à la compilation du wrapper
- Au sein d'un module, on peut desactiver director :

```
%feature(“nodirector”) classe;
```

Director
un exemple simple

```
%module(directors="1") exemple

%{
#include “classe.hpp”
%}

%feature(“director”) classe;

%include “classe.hpp”
```

Swig et STL

- Swig propose des fichiers d'interface pour la classe de la STL
- Utilisation :

  %include “std_vector.i”
  %include “std_map.i”
  %include “std_string.i”
- La commande %include permet d'inclure dans un fichier d'interface d'autres fichiers d'interface

Swig et les namespaces

- Swig sait analyser les namespaces
- on peut donc écrire, par exemple :
  %feature(“director”) n1::n2::classe; où n1 et n2 sont des namespaces
- Attention, il faudra inclure les namespaces comme les fichiers d'entête pour que la compilation se passe bien :

```
%{
using namespace n1::n2;
%}
```

La surcharge d'opérateurs

- swig génère automatiquement les wrappers sur les opérateurs surchargés (+, -, ...)
- dans certains cas (et dans certains langages) : impossibilités ou ambiguités
  - utilisation de la commande %rename
  - exemple :

```
%rename(add) operator+(double, const Type&)
```

Swig et Python

- Ligne de commande :
  - swig -c++ -python -nothreads example.i
  - deux fichiers sont générés :
    - le wrapper C++ : example_wrap.cxx
    - le wrapper Python : example.py
- Compilation du wrapper C++ :
  - g++ -c example.cpp -o example.o
  - g++ -I/usr/include/python2.5 -c example_wrap.cxx -o example_wrap.o
  - g++ -shared -fpic -o _example.so example.o example_wrap.o

Swig et CMake

- Il existe plusieurs possibilités :
  - la macro FindSwig
  - l'utilisation de la commande FIND_PACKAGE (solution présentée sur le site de Swig)
    - définition de la variable SWIG_USE_FILE contenant les fichiers d'inclusion de Swig
    - deux commandes :
      - SWIG_ADD_MODULE : définition du wrapper à générer
      - SWIG_LINK_LIBRARIES : définition des librairies nécessaires ; dépendant du langage cible

Swig et CMake
Un exemple en Python

- A mettre dans le CMakeLists.txt de la racine du projet
  - Recherche de Swig

```
FIND_PACKAGE(SWIG REQUIRED)
INCLUDE(${SWIG_USE_FILE})
```

  - Recherche de Python

```
INCLUDE(FindPythonLibs)
```

- A mettre dans le CMakeLists.txt où se trouve le fichier d'interface

```
INCLUDE_DIRECTORIES(${PYTHON_INCLUDE_PATH})
INCLUDE_DIRECTORIES(${CMAKE_CURRENT_SOURCE_DIR})
```

Swig et CMake
Un exemple en Python

- Définition des paramètres de swig
  - wrapper non threadé

```
SET(CMAKE_SWIG_FLAGS "-nothreads")
```

  - mode C++ (par défaut, mode C)

```
SET_SOURCE_FILES_PROPERTIES(example.i PROPERTIES
    CPLUSPLUS ON)
```

- Définition de la cible, du langage cible et des fichiers nécessaires :

```
SWIG_ADD_MODULE(example python example.i example.cpp)
```

- Définition des librairies nécessaires au linkage (ici, python) :

```
SWIG_LINK_LIBRARIES(example ${PYTHON_LIBRARIES})
```

- Les chemins d'installation

```
INSTALL(TARGETS _example DESTINATION "$
    {CMAKE_INSTALL_PREFIX}/lib/python/lib-dynload")
INSTALL(FILES ${CMAKE_SOURCE_DIR}/build/src/example.py
    DESTINATION "${CMAKE_INSTALL_PREFIX}/lib/python")
```