eBook Gratuit

APPRENEZ cmake

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#cmake

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: cmake

It is an unofficial and free cmake ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official cmake.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Commencer avec cmake

Remarques

CMake est un outil de définition et de gestion des versions de code, principalement pour C ++.

CMake est un outil multi-plateforme. L'idée est d'avoir une définition unique de la façon dont le projet est construit, ce qui se traduit par des définitions de construction spécifiques pour toute plate-forme prise en charge.

Cela se fait en associant différents systèmes de compilation spécifiques à la plate-forme; CMake est une étape intermédiaire qui génère des entrées de génération pour différentes plates-formes spécifiques. Sous Linux, CMake génère des Makefiles; sous Windows, il peut générer des projets Visual Studio, etc.

Le comportement de `CMakeLists.txt` est défini dans les fichiers `CMakeLists.txt` - un dans chaque répertoire du code source. Le fichier `CMakeLists` de `CMakeLists` répertoire définit ce que le buildsystem doit faire dans ce répertoire spécifique. Il définit également les sous-répertoires que CMake doit également gérer.

Les actions typiques incluent:

- Construisez une bibliothèque ou un exécutable à partir de certains des fichiers source de ce répertoire.
- Ajoutez un chemin de fichier au chemin d'inclusion utilisé lors de la génération.
- Définissez les variables que le buildsystem utilisera dans ce répertoire et dans ses sous-répertoires.
- Générez un fichier en fonction de la configuration de construction spécifique.
- Recherchez une bibliothèque qui se trouve quelque part dans l'arborescence source.

Les fichiers `CMakeLists` finaux peuvent être très clairs et simples, car leur portée est tellement limitée. Chacun ne gère que la quantité de construction présente dans le répertoire en cours.

Pour les ressources officielles sur CMake, consultez la documentation et le didacticiel de CMake.

Versions

Version	Date de sortie
3.9	2017-07-18
3.8	2017-04-10
3.7	2016-11-11
3.6	2016-07-07

Version	Date de sortie
3.5	2016-03-08
3.4	2015-11-12
3.3	2015-07-23
3.2	2015-03-10
3.1	2014-12-17
3.0	2014-06-10
2.8.12.1	2013-11-08
2.8.12	2013-10-11
2.8.11	2013-05-16
2.8.10.2	2012-11-27
2.8.10.1	2012-11-07
2.8.10	2012-10-31
2.8.9	2012-08-09
2.8.8	2012-04-18
2.8.7	2011-12-30
2.8.6	2011-12-30
2.8.5	2011-07-08
2.8.4	2011-02-16
2.8.3	2010-11-03
2.8.2	2010-06-28
2.8.1	2010-03-17
2.8	2009-11-13
2.6	2008-05-05

Examples

Installation CMake

Rendez -vous sur la page de téléchargement de CMake et obtenez un fichier binaire pour votre système d'exploitation, par exemple Windows, Linux ou Mac OS X. Sous Windows, double-cliquez sur le fichier binaire à installer. Sous Linux, lancez le binaire depuis un terminal.

Sous Linux, vous pouvez également installer les packages à partir du gestionnaire de packages de la distribution. Sur Ubuntu 16.04, vous pouvez installer l'application de ligne de commande et graphique avec:

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

Sur FreeBSD, vous pouvez installer la ligne de commande et l'application graphique basée sur Qt avec:

```
pkg install cmake
pkg install cmake-gui
```

Sur Mac OSX, si vous utilisez l'un des gestionnaires de paquets disponibles pour installer votre logiciel, le plus notable étant MacPorts (MacPorts) et Homebrew (Homebrew), vous pouvez également installer CMake via l'un d'eux. Par exemple, dans le cas de MacPorts, tapez ce qui suit

```
sudo port install cmake
```

va installer CMake, tandis que si vous utilisez le gestionnaire de paquets Homebrew, vous devrez taper

```
brew install cmake
```

Une fois que vous avez installé CMake, vous pouvez vérifier facilement en procédant comme suit

```
cmake --version
```

Vous devriez voir quelque chose de similaire à ce qui suit

```
cmake version 3.5.1

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

Basculer entre les types de construction, par exemple le débogage et la libération

CMake connaît plusieurs types de construction, qui influencent généralement les paramètres par défaut du compilateur et de l'éditeur de liens (tels que les informations de débogage en cours de création) ou d'autres chemins de code.

Par défaut, CMake est capable de gérer les types de construction suivants:

- **Debug** : généralement une version de débogage classique incluant des informations de débogage, aucune optimisation, etc.
- **Version** : votre version standard sans informations de débogage et optimisation complète.
- **RelWithDebInfo::** Identique à *Release* , mais avec des informations de débogage.
- **MinSizeRel** : une *version* spéciale optimisée pour la taille.

La manière dont les configurations sont gérées dépend du générateur utilisé.

Certains générateurs (comme Visual Studio) prennent en charge plusieurs configurations. CMake générera toutes les configurations en même temps et vous pourrez sélectionner dans l'IDE ou utiliser `--config CONFIG` (avec `cmake --build`) quelle configuration vous voulez construire. Pour ces générateurs, CMake fera de son mieux pour générer une structure de répertoire de construction de sorte que les fichiers provenant de différentes configurations ne se chevauchent pas.

Les générateurs qui ne prennent en charge qu'une seule configuration (comme les Makefiles Unix) fonctionnent différemment. Ici, la configuration actuellement active est déterminée par la valeur de la variable CMake `CMAKE_BUILD_TYPE` .

Par exemple, pour choisir un type de construction différent, vous pouvez émettre les commandes de ligne de commande suivantes:

```
cmake -DCMAKE_BUILD_TYPE=Debug path/to/source
cmake -DCMAKE_BUILD_TYPE=Release path/to/source
```

Un script CMake devrait éviter de définir le `CMAKE_BUILD_TYPE` lui-même, car il est généralement considéré comme la responsabilité de l'utilisateur de le faire.

Pour les générateurs à configuration unique, la commutation de la configuration nécessite le redémarrage de CMake. Une génération ultérieure est susceptible de remplacer les fichiers objets produits par la configuration antérieure.

Projet simple "Hello World"

Étant donné qu'un fichier source C ++ `main.cpp` définissant une fonction `main()` , un fichier `CMakeLists.txt` (avec le contenu suivant) indiquera à *CMake* de générer les instructions de génération appropriées pour le système actuel et le compilateur C ++ par défaut.

main.cpp *(exemple Hello C ++)*

```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
    return 0;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.4)

project(hello_world)

add_executable(app main.cpp)
```

Voir en direct sur Coliru

1. `cmake_minimum_required(VERSION 2.4)` définit une version minimale de CMake requise pour évaluer le script en cours.

2. `project(hello_world)` lance un nouveau projet CMake. Cela déclenchera beaucoup de logique CMake interne, en particulier la détection du compilateur C et C ++ par défaut.

3. Avec `add_executable(app main.cpp)` une `app` cible de génération est créée, qui invoque le compilateur configuré avec quelques indicateurs par défaut pour le paramètre actuel afin de compiler une `app` exécutable à partir du fichier source donné `main.cpp` .

Ligne de commande *(In-Source-Build, non recommandé)*

```
> cmake .
...
> cmake --build .
```

`cmake .` fait la détection du compilateur, évalue le `CMakeLists.txt` dans le donné `.` répertoire et génère l'environnement de génération dans le répertoire de travail en cours.

Le `cmake --build .` commande est une abstraction pour la construction / appel nécessaire.

Ligne de commande *(hors source, recommandé)*

Pour que votre code source reste propre à tous les artefacts de build, vous devez effectuer des builds "out-of-source".

```
> mkdir build
> cd build
> cmake ..
> cmake --build .
```

Ou CMake peut également abstraire les commandes de base de votre shell de plate-forme à partir de l'exemple ci-dessus:

```
> cmake -E make_directory build
> cmake -E chdir build cmake ..
> cmake --build build
```

"Hello World" avec plusieurs fichiers sources

Tout d'abord, nous pouvons spécifier les répertoires des fichiers d'en-tête par `include_directories()` , puis nous devons spécifier les fichiers sources correspondants de l'exécutable cible par `add_executable()` , et nous devons être sûrs qu'il existe une fonction `main()`

dans les fichiers source.

Voici un exemple simple, tous les fichiers sont supposés placés dans le répertoire `PROJECT_SOURCE_DIR` .

main.cpp

```
#include "foo.h"

int main()
{
    foo();
    return 0;
}
```

foo.h

```
void foo();
```

foo.cpp

```
#include <iostream>
#include "foo.h"

void foo()
{
    std::cout << "Hello World!\n";
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.4)

project(hello_world)

include_directories(${PROJECT_SOURCE_DIR})
add_executable(app main.cpp foo.cpp)  # be sure there's exactly one main() function in the
source files
```

Nous pouvons suivre la même procédure dans l' exemple ci - dessus pour construire notre projet. Ensuite, l' `app` exécution imprimera

```
>./app
Hello World!
```

"Hello World" en bibliothèque

Cet exemple montre comment déployer le programme "Hello World" en tant que bibliothèque et comment le lier aux autres cibles.

Disons que nous avons le même ensemble de fichiers source / en-tête que dans l'exemple http://www.riptutorial.com/cmake/example/22391/-hello-world--with-multiple-source-files . Au lieu

de construire à partir de plusieurs fichiers sources, nous pouvons d'abord déployer `foo.cpp` tant que bibliothèque en utilisant `add_library()` et en le liant ensuite au programme principal avec `target_link_libraries()` .

Nous modifions **CMakeLists.txt** en

```
cmake_minimum_required(VERSION 2.4)

project(hello_world)

include_directories(${PROJECT_SOURCE_DIR})
add_library(applib foo.cpp)
add_executable(app main.cpp)
target_link_libraries(app applib)
```

et en suivant les mêmes étapes, nous obtiendrons le même résultat.

Lire Commencer avec cmake en ligne: https://riptutorial.com/fr/cmake/topic/862/commencer-avec-cmake

Chapitre 2: Ajouter des répertoires au chemin d'inclusion du compilateur

Syntaxe

- include_directories ([APRÈS | AVANT] [SYSTÈME] dir1 [dir2 ...])

Paramètres

Paramètre	La description
`dirN`	un ou plusieurs chemins relatifs ou absolus
`AFTER` , `BEFORE`	(facultatif) si les répertoires donnés doivent être ajoutés au début ou à la fin de la liste actuelle des chemins d'inclusion; le comportement par défaut est défini par `CMAKE_INCLUDE_DIRECTORIES_BEFORE`
`SYSTEM`	(optionnel) indique au compilateur de suivre les répertoires donnés en tant que *système incluant les* répertoires, ce qui peut déclencher une gestion spéciale par le compilateur

Examples

Ajouter un sous-répertoire de projet

Compte tenu de la structure de projet suivante

```
include\
  myHeader.h
src\
  main.cpp
CMakeLists.txt
```

la ligne suivante dans le fichier `CMakeLists.txt`

```
include_directories(${PROJECT_SOURCE_DIR}/include)
```

Permet d' ajouter le `include` répertoire au *chemin de recherche* du compilateur pour toutes les cibles définies dans ce répertoire (et tous ses sous - répertoires inclus via `add_subdirectory()`).

Ainsi, le fichier `myHeader.h` du sous `include` répertoire `include` du projet peut être inclus via `#include "myHeader.h"` dans le fichier `main.cpp` .

Lire Ajouter des répertoires au chemin d'inclusion du compilateur en ligne:

https://riptutorial.com/fr/cmake/topic/5968/ajouter-des-repertoires-au-chemin-d-inclusion-du-compilateur

Chapitre 3: Configurations de construction

Introduction

Cette rubrique présente les utilisations de différentes configurations CMake telles que Debug ou Release, dans différents environnements.

Examples

Définition d'une configuration Release / Debug

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.8.11)
SET(PROJ_NAME "myproject")
PROJECT(${PROJ_NAME})

# Configuration types
SET(CMAKE_CONFIGURATION_TYPES "Debug;Release" CACHE STRING "Configs" FORCE)
IF(DEFINED CMAKE_BUILD_TYPE AND CMAKE_VERSION VERSION_GREATER "2.8")
  SET_PROPERTY(CACHE CMAKE_BUILD_TYPE PROPERTY STRINGS  ${CMAKE_CONFIGURATION_TYPES})
ENDIF()

SET(${PROJ_NAME}_PATH_INSTALL     "/opt/project"                    CACHE PATH "This
directory contains installation Path")
SET(CMAKE_DEBUG_POSTFIX "d")

# Install
#-----------------------------------------------------#
INSTALL(TARGETS ${PROJ_NAME}
    DESTINATION  "${${PROJ_NAME}_PATH_INSTALL}/lib/${CMAKE_BUILD_TYPE}/"
    )
```

Effectuer les versions suivantes générera deux dossiers différents ('/ opt / myproject / lib / Debug' '/ opt / myproject / lib / Release') avec les bibliothèques:

```
$ cd /myproject/build
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
$ make
$ sudo make install
$ cmake _DCMAKE_BUILD_TYPE=Release ..
$ make
$ sudo make install
```

Lire Configurations de construction en ligne:
https://riptutorial.com/fr/cmake/topic/8319/configurations-de-construction

Chapitre 4: Configurer le fichier

Introduction

configure_file est une fonction CMake permettant de copier un fichier vers un autre emplacement et de modifier son contenu. Cette fonction est très utile pour générer des fichiers de configuration avec des chemins, des variables personnalisées, en utilisant un modèle générique.

Remarques

Copiez un fichier vers un autre emplacement et modifiez son contenu.

```
configure_file(<input> <output>
           [COPYONLY] [ESCAPE_QUOTES] [@ONLY]
           [NEWLINE_STYLE [UNIX|DOS|WIN32|LF|CRLF] ])
```

Copie un fichier pour classer et remplacer les valeurs de variable référencées dans le contenu du fichier. Si est un chemin relatif, il est évalué par rapport au répertoire source actuel. Le doit être un fichier, pas un répertoire. Si est un chemin relatif, il est évalué par rapport au répertoire binaire actuel. Si nomme un répertoire existant, le fichier d'entrée est placé dans ce répertoire avec son nom d'origine.

Si le fichier est modifié, le système de génération réexécutera CMake pour reconfigurer le fichier et générer à nouveau le système de génération.

Cette commande remplace toutes les variables du fichier d'entrée référencées $ {VAR} ou @ VAR @ par leurs valeurs déterminées par CMake. Si une variable n'est pas définie, elle sera remplacée par rien. Si COPYONLY est spécifié, aucune expansion de variable n'aura lieu. Si ESCAPE_QUOTES est spécifié, toutes les citations substituées seront échappées avec le style C. Le fichier sera configuré avec les valeurs actuelles des variables CMake. Si @ONLY est spécifié, seules les variables de la forme @ VAR @ seront remplacées et $ {VAR} sera ignoré. Ceci est utile pour configurer des scripts qui utilisent $ {VAR}.

Les lignes de fichier d'entrée de la forme "#cmakedefine VAR ..." seront remplacées par "#define VAR ..." ou / * #undef VAR * / selon que VAR est défini dans CMake à une valeur non considérée comme fausse constante par la commande if (). (Le contenu de "...", le cas échéant, est traité comme ci-dessus.) Les lignes de fichier d'entrée de la forme "# cmakedefine01 VAR" seront remplacées de manière similaire par "#define VAR 1" ou "#define VAR 0".

Avec NEWLINE_STYLE, la fin de ligne peut être ajustée:

```
'UNIX' or 'LF' for \n, 'DOS', 'WIN32' or 'CRLF' for \r\n.
```

COPYONLY ne doit pas être utilisé avec NEWLINE_STYLE.

Examples

Générer un fichier de configuration c ++ avec CMake

Si nous avons un projet c ++ qui utilise un fichier de configuration config.h avec des chemins ou des variables personnalisés, nous pouvons le générer à l'aide de CMake et d'un fichier générique config.h.in.

Le fichier config.h.in peut faire partie d'un dépôt git, tandis que le fichier config.h généré ne sera jamais ajouté, car il est généré à partir de l'environnement actuel.

```
#CMakeLists.txt
CMAKE_MINIMUM_REQUIRED(VERSION 2.8.11)

SET(PROJ_NAME "myproject")
PROJECT(${PROJ_NAME})

SET(${PROJ_NAME}_DATA     ""     CACHE PATH "This directory contains all DATA and RESOURCES")
SET(THIRDPARTIES_PATH    ${CMAKE_CURRENT_SOURCE_DIR}/../thirdparties      CACHE PATH "This
directory contains thirdparties")

configure_file ("${CMAKE_CURRENT_SOURCE_DIR}/common/config.h.in"
            "${CMAKE_CURRENT_SOURCE_DIR}/include/config.h" )
```

Si nous avons un config.h.in comme ceci:

```
cmakedefine PATH_DATA "@myproject_DATA@"
cmakedefine THIRDPARTIES_PATH "@THIRDPARTIES_PATH@"
```

Les précédentes CMakeLists génèrent un en-tête c ++ comme ceci:

```
#define PATH_DATA "/home/user/projects/myproject/data"
#define THIRDPARTIES_PATH "/home/user/projects/myproject/thirdparties"
```

Examen basé sur la version de contrôle SDL2

Si vous avez un module `cmake` . Vous pouvez créer un dossier appelé `in` pour stocker tous les fichiers de configuration.

Par exemple, vous avez un projet appelé `FOO` , vous pouvez créer un fichier `FOO_config.h.in` comme:

```
//=============================================================================
//  CMake configuration file, based on SDL 2 version header
// ============================================================================

#pragma once

#include <string>
#include <sstream>
```

```
namespace yournamespace
{
  /**
 *  \brief Information the version of FOO_PROJECT in use.
 *
 *  Represents the library's version as three levels: major revision
 *  (increments with massive changes, additions, and enhancements),
 *  minor revision (increments with backwards-compatible changes to the
 *  major revision), and patchlevel (increments with fixes to the minor
 *  revision).
 *
 *  \sa FOO_VERSION
 *  \sa FOO_GetVersion
 */
typedef struct FOO_version
{
    int major;        /**< major version */
    int minor;        /**< minor version */
    int patch;        /**< update version */
} FOO_version;

/* Printable format: "%d.%d.%d", MAJOR, MINOR, PATCHLEVEL
*/
#define FOO_MAJOR_VERSION   0
#define FOO_MINOR_VERSION   1
#define FOO_PATCHLEVEL      0

/**
 *  \brief Macro to determine FOO version program was compiled against.
 *
 *  This macro fills in a FOO_version structure with the version of the
 *  library you compiled against. This is determined by what header the
 *  compiler uses. Note that if you dynamically linked the library, you might
 *  have a slightly newer or older version at runtime. That version can be
 *  determined with GUCpp_GetVersion(), which, unlike GUCpp_VERSION(),
 *  is not a macro.
 *
 *  \param x A pointer to a FOO_version struct to initialize.
 *
 *  \sa FOO_version
 *  \sa FOO_GetVersion
 */
#define FOO_VERSION(x)                          \
{                                       \
    (x)->major = FOO_MAJOR_VERSION;                 \
    (x)->minor = FOO_MINOR_VERSION;                 \
    (x)->patch = FOO_PATCHLEVEL;                    \
}

/**
 *  This macro turns the version numbers into a numeric value:
 *  \verbatim
    (1,2,3) -> (1203)
    \endverbatim
 *
 *  This assumes that there will never be more than 100 patchlevels.
 */
#define FOO_VERSIONNUM(X, Y, Z)                     \
    ((X)*1000 + (Y)*100 + (Z))

/**
```

```
 *  This is the version number macro for the current GUCpp version.
 */
#define FOO_COMPILEDVERSION \
    FOO_VERSIONNUM(FOO_MAJOR_VERSION, FOO_MINOR_VERSION, FOO_PATCHLEVEL)

/**
 *  This macro will evaluate to true if compiled with FOO at least X.Y.Z.
 */
#define FOO_VERSION_ATLEAST(X, Y, Z) \
    (FOO_COMPILEDVERSION >= FOO_VERSIONNUM(X, Y, Z))

}

// Paths
#cmakedefine FOO_PATH_MAIN "@FOO_PATH_MAIN@"
```

Ce fichier créera un `FOO_config.h` dans le chemin d'installation, avec une variable définie dans c `FOO_PATH_MAIN` partir de la variable cmake. Pour le générer, vous devez inclure `in` fichier dans votre CMakeLists.txt, comme ceci (définir des chemins et des variables):

```
MESSAGE("Configuring FOO_config.h ...")
configure_file("${CMAKE_CURRENT_SOURCE_DIR}/common/in/FOO_config.h.in"
"${FOO_PATH_INSTALL}/common/include/FOO_config.h" )
```

Ce fichier contiendra les données du modèle, et variable avec votre chemin réel, par exemple:

```
// Paths
#define FOO_PATH_MAIN "/home/YOUR_USER/Respositories/git/foo_project"
```

Lire Configurer le fichier en ligne: https://riptutorial.com/fr/cmake/topic/8304/configurer-le-fichier

Chapitre 5: Construire des cibles

Syntaxe

- add_executable (target_name [EXCLUDE_FROM_ALL] source1 [source2 ...])
- add_library (lib_name [STATIC | SHARED | MODULE] [EXCLUDE_FROM_ALL] source1 [source2 ...])

Examples

Des exécutables

Pour créer une cible de génération produisant un exécutable, il faut utiliser la commande `add_executable` :

```
add_executable(my_exe
               main.cpp
               utilities.cpp)
```

Cela crée une cible de génération, par exemple `make my_exe` pour GNU make, avec les `make my_exe` appropriés du compilateur configuré pour produire un exécutable `my_exe` partir des deux fichiers sources `main.cpp` et `utilities.cpp` .

Par défaut, toutes les cibles exécutables sont ajoutées à `all` cibles intégrées (`all` pour GNU make, `BUILD_ALL` pour `BUILD_ALL`).
Pour exclure un exécutable d'être construit avec la valeur par défaut `all` les cibles, on peut ajouter le paramètre optionnel `EXCLUDE_FROM_ALL` juste après le nom de la cible:

```
add_executable(my_optional_exe EXCLUDE_FROM_ALL main.cpp)
```

Bibliothèques

Pour créer une cible de génération qui crée une bibliothèque, utilisez la commande `add_library` :

```
add_library(my_lib lib.cpp)
```

La variable CMake `BUILD_SHARED_LIBS` contrôle à quel moment créer une bibliothèque statique (`OFF`) ou partagée (`ON`), en utilisant par exemple `cmake .. -DBUILD_SHARED_LIBS=ON` . Cependant, vous pouvez explicitement définir une bibliothèque partagée ou statique en ajoutant `STATIC` ou `SHARED` après le nom de la cible:

```
add_library(my_shared_lib SHARED lib.cpp) # Builds an shared library
add_library(my_static_lib STATIC lib.cpp) # Builds an static library
```

Le fichier de sortie réel diffère entre les systèmes. Par exemple, une bibliothèque partagée sur les

systèmes Unix est généralement appelée `libmy_shared_library.so` , mais sous Windows, il s'agirait de `my_shared_library.dll` et de `my_shared_library.lib` .

Comme `add_executable` , ajoutez `EXCLUDE_FROM_ALL` avant la liste des fichiers sources pour l'exclure de la cible `all` :

```
add_library(my_lib EXCLUDE_FROM_ALL lib.cpp)
```

Les bibliothèques conçues pour être chargées à l'exécution (par exemple des plugins ou des applications utilisant quelque chose comme `dlopen`) doivent utiliser `MODULE` au lieu de `SHARED` / `STATIC` :

```
add_library(my_module_lib MODULE lib.cpp)
```

Par exemple, sous Windows, il n'y aura pas de fichier d'importation (`.lib`), car les symboles sont directement exportés dans le fichier `.dll` .

Lire Construire des cibles en ligne: https://riptutorial.com/fr/cmake/topic/3107/construire-des-cibles

Chapitre 6: Créer des suites de test avec CTest

Examples

Suite de tests de base

```
# the usual boilerplate setup
cmake_minimum_required(2.8)
project(my_test_project
        LANGUAGES CXX)

# tell CMake to use CTest extension
enable_testing()

# create an executable, which instantiates a runner from
# GoogleTest, Boost.Test, QtTest or whatever framework you use
add_executable(my_test
               test_main.cpp)

# depending on the framework, you need to link to it
target_link_libraries(my_test
                      gtest_main)

# now register the executable with CTest
add_test(NAME my_test COMMAND my_test)
```

La macro `enable_testing()` fait beaucoup de magie. Tout d'abord, il crée une cible builtin `test` (pour GNU make, `RUN_TESTS` pour VS), qui, lors de son exécution, exécute *CTest.*

L'appel à `add_test()` enregistre finalement un exécutable arbitraire avec *CTest* , ainsi l'exécutable est exécuté à chaque fois que nous appelons la cible de `test` .

Maintenant, construisez le projet comme d'habitude et enfin exécutez la cible de test

GNU Make	Visual Studio
`make test`	`cmake --build . --target RUN_TESTS`

Lire Créer des suites de test avec CTest en ligne: https://riptutorial.com/fr/cmake/topic/4197/creer-des-suites-de-test-avec-ctest

Chapitre 7: Étapes de construction personnalisées

Introduction

Les étapes de construction personnalisées sont utiles pour exécuter des cibles personnalisées dans la construction de votre projet ou pour copier facilement des fichiers afin de ne pas avoir à le faire manuellement (peut-être des DLL?). Ici, je vais vous montrer deux exemples, le premier est de copier les DLL (en particulier QLL5 DLL) dans le répertoire binaire de vos projets (Debug ou Release) et le second est de lancer une cible personnalisée (Doxygen dans ce cas) dans votre solution. (si vous utilisez Visual Studio).

Remarques

Comme vous pouvez le voir, vous pouvez faire beaucoup de choses avec les cibles et étapes de construction personnalisées dans cmake, mais vous devez être prudent lorsque vous les utilisez, en particulier lorsque vous copiez des DLL. Bien qu'il soit utile de le faire, cela peut parfois conduire à ce que l'on appelle affectueusement «Dll Hell».

Fondamentalement, cela signifie que vous pouvez vous perdre dans les DLL dont dépend réellement votre exécutable, lesquelles sont chargées et quelles sont celles qui doivent être exécutées (peut-être à cause de la variable de chemin de votre ordinateur).

Hormis ce qui précède, n'hésitez pas à faire en sorte que les cibles personnalisées fassent ce que vous voulez! Ils sont puissants et flexibles et constituent un outil précieux pour tout projet de cmake.

Examples

Qt5 dll copier exemple

Donc, disons que vous avez un projet qui dépend de Qt5 et que vous devez copier les DLL appropriées dans votre répertoire de construction et que vous ne voulez pas le faire manuellement; vous pouvez faire ce qui suit:

```
cmake_minimum_required(VERSION 3.0)
project(MyQtProj LANGUAGES C CXX)
find_package(Qt5 COMPONENTS Core Gui Widgets)
#...set up your project

# add the executable
add_executable(MyQtProj ${PROJ_SOURCES} ${PROJ_HEADERS})

add_custom_command(TARGET MyQtProj POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Core>
$<TARGET_FILE_DIR:MyQtProj>
```

```
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Gui>
$<TARGET_FILE_DIR:MyQtProj>
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Widgets>
$<TARGET_FILE_DIR:MyQtProj>
    )
```

Donc, chaque fois que vous construisez votre projet, si les DLL cibles ont été modifiées et que vous souhaitez les copier, elles seront copiées après la construction de votre cible (ici le fichier exécutable principal) (notez la commande `copy_if_different`); sinon, ils ne seront pas copiés.

De plus, notez l'utilisation des expressions de générateur ici. L'avantage de leur utilisation est que vous n'avez pas à dire explicitement où copier les DLL ou quelles variantes utiliser. Pour pouvoir les utiliser, le projet que vous utilisez (Qt5 dans ce cas) doit avoir des cibles importées.

Si vous construisez dans le débogage, CMake sait alors (en fonction de la cible importée) copier les fichiers Qt5Cored.dll, Qt5Guid.dll et Qt5Widgetsd.dll dans le dossier Debug de votre dossier de génération. Si vous construisez dans la version, les versions de publication des fichiers .dll seront copiées dans le dossier de la version.

Exécution d'une cible personnalisée

Vous pouvez également créer une cible personnalisée à exécuter lorsque vous souhaitez effectuer une tâche particulière. Ce sont généralement des exécutables que vous exécutez pour faire des choses différentes. Quelque chose qui peut être particulièrement utile est d'exécuter Doxygen pour générer de la documentation pour votre projet. Pour ce faire, vous pouvez effectuer les opérations suivantes dans votre `CMakeLists.txt` (par souci de simplicité, nous continuerons notre exemple de projet Qt5):

```
cmake_minimum_required(VERSION 3.0)
project(MyQtProj LANGUAGES C CXX)
find_package(Qt5 COMPONENTS Core Gui Widgets)
#...set up your project

add_executable(MyQtProj ${PROJ_SOURCES} ${PROJ_HEADERS})

add_custom_command(TARGET MyQtProj POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Core>
$<TARGET_FILE_DIR:MyQtProj>
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Gui>
$<TARGET_FILE_DIR:MyQtProj>
    COMMAND ${CMAKE_COMMAND} -E copy_if_different $<TARGET_FILE:Qt5::Widgets>
$<TARGET_FILE_DIR:MyQtProj>
    )

#Add target to build documents from visual studio.
set(DOXYGEN_INPUT ${CMAKE_CURRENT_SOURCE_DIR}/Doxyfile)
#set the output directory of the documentation
set(DOXYGEN_OUTPUT_DIR ${CMAKE_CURRENT_SOURCE_DIR}/docs)
# sanity check...
message("Doxygen Output ${DOXYGEN_OUTPUT_DIR}")
find_package(Doxygen)

if(DOXYGEN_FOUND)
    # create the output directory where the documentation will live
```

```
    file(MAKE_DIRECTORY ${DOXYGEN_OUTPUT_DIR})
    # configure our Doxygen configuration file. This will be the input to the doxygen
    # executable
    configure_file(${CMAKE_CURRENT_SOURCE_DIR}/Doxyfile.in
    ${CMAKE_CURRENT_BINARY_DIR}/Doxyfile @ONLY)

# now add the custom target. This will create a build target called 'DOCUMENTATION'
# in your project
ADD_CUSTOM_TARGET(DOCUMENTATION
  COMMAND ${CMAKE_COMMAND} -E echo_append "Building API Documentation..."
  COMMAND ${CMAKE_COMMAND} -E make_directory ${DOXYGEN_OUTPUT_DIR}
  COMMAND ${DOXYGEN_EXECUTABLE} ${CMAKE_CURRENT_BINARY_DIR}/Doxyfile
    WORKING_DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}
  COMMAND ${CMAKE_COMMAND} -E echo "Done."
  WORKING_DIRECTORY ${DOXYGEN_OUTPUT_DIR})

endif(DOXYGEN_FOUND)
```

Maintenant, lorsque nous créons notre solution (en supposant que vous utilisez Visual Studio), vous aurez une cible de génération appelée `DOCUMENTATION` que vous pouvez créer pour régénérer la documentation de votre projet.

Lire Étapes de construction personnalisées en ligne:
https://riptutorial.com/fr/cmake/topic/9537/etapes-de-construction-personnalisees

Chapitre 8: Fonctions de compilation et sélection standard C / C ++

Syntaxe

- target_compile_features *(cible* **PRIVÉ | PUBLIC | INTERFACE** *feature1 [feature2* ...])

Examples

Compiler les conditions requises

Les fonctions de compilation requises peuvent être spécifiées sur une cible à l'aide de la commande target_compile_features :

```
add_library(foo
    foo.cpp
)
target_compile_features(foo
    PRIVATE          # scope of the feature
    cxx_constexpr    # list of features
)
```

Les fonctionnalités doivent faire partie de CMAKE_C_COMPILE_FEATURES ou CMAKE_CXX_COMPILE_FEATURES ; cmake signale une erreur sinon. Cmake ajoutera les options nécessaires telles que `-std=gnu++11` aux options de compilation de la cible.

Dans l'exemple, les fonctionnalités sont déclarées `PRIVATE` : les exigences seront ajoutées à la cible, mais pas à ses consommateurs. Pour ajouter automatiquement les exigences à un bâtiment cible par rapport à foo, vous devez utiliser `PUBLIC` ou `INTERFACE` au lieu de `PRIVATE` :

```
target_compile_features(foo
    PUBLIC    # this time, required as public
    cxx_constexpr
)

add_executable(bar
    main.cpp
)
target_link_libraries(bar
    foo       # foo's public requirements and compile flags are added to bar
)
```

Sélection de version C / C ++

Les versions `CMAKE_C_STANDARD` pour C et C ++ peuvent être spécifiées globalement en utilisant respectivement les variables `CMAKE_C_STANDARD` (les valeurs acceptées sont 98, 99 et 11) et `CMAKE_CXX_STANDARD` (les valeurs acceptées sont 98, 11 et 14):

```
set(CMAKE_C_STANDARD 99)
set(CMAKE_CXX_STANDARD 11)
```

Celles-ci ajouteront les options de compilation nécessaires sur les cibles (par exemple, `-std=c++11` pour gcc).

La version peut être rendue obligatoire en définissant sur `ON` les variables `CMAKE_C_STANDARD_REQUIRED` et `CMAKE_CXX_STANDARD_REQUIRED` respectivement.

Les variables doivent être définies avant la création de la cible. La version peut également être spécifiée par cible:

```
set_target_properties(foo PROPERTIES
    CXX_STANDARD 11
    CXX_STANDARD_REQUIRED ON
)
```

Lire Fonctions de compilation et sélection standard C / C ++ en ligne: https://riptutorial.com/fr/cmake/topic/5297/fonctions-de-compilation-et-selection-standard-c---c-plusplus

Chapitre 9: Fonctions et macros

Remarques

La principale différence entre les *macros* et les *fonctions* est que les *macros* sont évaluées dans le contexte actuel, tandis que les *fonctions* ouvrent une nouvelle étendue au sein du contexte actuel. Ainsi, les variables définies dans les *fonctions* ne sont pas connues après l'évaluation de la fonction. Au contraire, les variables au sein des *macros* sont toujours définies après évaluation de la macro.

Examples

Macro simple pour définir une variable en fonction de l'entrée

```
macro(set_my_variable _INPUT)
  if("${_INPUT}" STREQUAL "Foo")
    set(my_output_variable "foo")
  else()
    set(my_output_variable "bar")
  endif()
endmacro(set_my_variable)
```

Utilisez la macro:

```
set_my_variable("Foo")
message(STATUS ${my_output_variable})
```

imprimera

```
-- foo
```

tandis que

```
set_my_variable("something else")
message(STATUS ${my_output_variable})
```

imprimera

```
-- bar
```

Macro pour remplir une variable de prénom

```
macro(set_custom_variable _OUT_VAR)
  set(${_OUT_VAR} "Foo")
endmacro(set_custom_variable)
```

Utilisez-le avec

```
set_custom_variable(my_foo)
message(STATUS ${my_foo})
```

qui imprimera

```
-- Foo
```

Lire Fonctions et macros en ligne: https://riptutorial.com/fr/cmake/topic/2096/fonctions-et-macros

Chapitre 10: Intégration CMake dans les outils GitHub CI

Examples

Configurer Travis CI avec stock CMake

Travis CI a CMake 2.8.7 pré-installé.

Un script minimal `.travis.yml` pour une version source

```
language: cpp

compiler:
  - gcc

before_script:
  # create a build folder for the out-of-source build
  - mkdir build
  # switch to build directory
  - cd build
  # run cmake; here we assume that the project's
  # top-level CMakeLists.txt is located at '..'
  - cmake ..

script:
  # once CMake has done its job we just build using make as usual
  - make
  # if the project uses ctest we can run the tests like this
  - make test
```

Configurez Travis CI avec le dernier CMake

La version CMake préinstallée sur Travis est très ancienne. Vous pouvez utiliser les binaires Linux officiels pour construire avec une version plus récente.

Voici un exemple `.travis.yml` :

```
language: cpp

compiler:
  - gcc

# the install step will take care of deploying a newer cmake version
install:
  # first we create a directory for the CMake binaries
  - DEPS_DIR="${TRAVIS_BUILD_DIR}/deps"
  - mkdir ${DEPS_DIR} && cd ${DEPS_DIR}
  # we use wget to fetch the cmake binaries
  - travis_retry wget --no-check-certificate https://cmake.org/files/v3.3/cmake-3.3.2-Linux-
x86_64.tar.gz
```

```
  # this is optional, but useful:
  # do a quick md5 check to ensure that the archive we downloaded did not get compromised
  - echo "f3546812c11ce7f5d64dc132a566b749 *cmake-3.3.2-Linux-x86_64.tar.gz" > cmake_md5.txt
  - md5sum -c cmake_md5.txt
  # extract the binaries; the output here is quite lengthy,
  # so we swallow it to not clutter up the travis console
  - tar -xvf cmake-3.3.2-Linux-x86_64.tar.gz > /dev/null
  - mv cmake-3.3.2-Linux-x86_64 cmake-install
  # add both the top-level directory and the bin directory from the archive
  # to the system PATH. By adding it to the front of the path we hide the
  # preinstalled CMake with our own.
  - PATH=${DEPS_DIR}/cmake-install:${DEPS_DIR}/cmake-install/bin:$PATH
  # don't forget to switch back to the main build directory once you are done
  - cd ${TRAVIS_BUILD_DIR}

before_script:
  # create a build folder for the out-of-source build
  - mkdir build
  # switch to build directory
  - cd build
  # run cmake; here we assume that the project's
  # top-level CMakeLists.txt is located at '..'
  - cmake ..

script:
  # once CMake has done its job we just build using make as usual
  - make
  # if the project uses ctest we can run the tests like this
  - make test
```

Lire Intégration CMake dans les outils GitHub CI en ligne:
https://riptutorial.com/fr/cmake/topic/1445/integration-cmake-dans-les-outils-github-ci

Chapitre 11: Projet hiérarchique

Examples

Approche simple sans paquets

Exemple qui construit un exécutable (éditeur) et lui associe une bibliothèque (mettre en évidence). La structure du projet est simple, il faut un maître CMakeLists et un répertoire pour chaque sous-projet:

```
CMakeLists.txt
editor/
    CMakeLists.txt
    src/
        editor.cpp
highlight/
    CMakeLists.txt
    include/
        highlight.h
    src/
        highlight.cpp
```

Le fichier CMakeLists.txt principal contient des définitions globales et `add_subdirectory` appel `add_subdirectory` pour chaque sous-projet:

```
cmake_minimum_required(VERSION 3.0)
project(Example)

add_subdirectory(highlight)
add_subdirectory(editor)
```

CMakeLists.txt pour la bibliothèque lui affecte des sources et y inclut des répertoires. En utilisant `target_include_directories()` au lieu de `include_directories()` les répertoires include seront propagés aux utilisateurs de la bibliothèque:

```
cmake_minimum_required(VERSION 3.0)
project(highlight)

add_library(${PROJECT_NAME} src/highlight.cpp)
target_include_directories(${PROJECT_NAME} PUBLIC include)
```

CMakeLists.txt pour l'application attribue des sources et lie la bibliothèque de surbrillance. Les chemins d'accès au binaire de hightlighter et aux include sont gérés automatiquement par cmake:

```
cmake_minimum_required(VERSION 3.0)
project(editor)

add_executable(${PROJECT_NAME} src/editor.cpp)
target_link_libraries(${PROJECT_NAME} PUBLIC highlight)
```

Lire Projet hiérarchique en ligne: https://riptutorial.com/fr/cmake/topic/1443/projet-hierarchique

Chapitre 12: Projets d'emballage et de distribution

Syntaxe

- # Package d'un répertoire de construction
  pack [CHEMIN]
- # Utiliser un générateur spécifique
  cpack -G [GENERATEUR] [CHEMIN]
- # Fournit des remplacements facultatifs
- cpack -G [GENERATEUR] -C [CONFIGURATION] -P [NOM DU PAQUET] -R [VERSION DU PAQUET] -B [ANNUAIRE DU PAQUET] --vendeur [VENDEUR PAQUET]

Remarques

CPack est un outil externe permettant de conditionner rapidement les projets CMake construits en rassemblant toutes les données requises directement à partir des fichiers `CMakeLists.txt` et des commandes d'installation utilisées telles que `install_targets()` .

Pour que CPack fonctionne correctement, le `CMakeLists.txt` doit inclure des fichiers ou des cibles à installer à l'aide de la cible de génération d' `install` .

Un script minimal pourrait ressembler à ceci:

```
# Required headers
cmake(3.0)

# Basic project setup
project(my-tool)

# Define a buildable target
add_executable(tool main.cpp)

# Provide installation instructions
install_targets(tool DESTINATION bin)
```

Examples

Créer un paquet pour un projet CMake construit

Pour créer un package redistribuable (par exemple une archive ZIP ou un programme d'installation), il suffit généralement d'appeler simplement CPack en utilisant une syntaxe très similaire à l'appel de CMake:

```
cpack path/to/build/directory
```

Selon l'environnement, cela rassemblera tous les fichiers requis / installés pour le projet et les placera dans une archive compressée ou un programme d'installation auto-extractible.

Sélection d'un générateur CPack à utiliser

Pour créer un package en utilisant un format spécifique, il est possible de choisir le **générateur** à utiliser.

Similaire à CMake, cela peut être fait en utilisant l'argument **-G** :

```
cpack -G 7Z .
```

L'utilisation de cette ligne de commande empaquera le projet intégré dans le répertoire en cours en utilisant le format d'archive 7-Zip.

Au moment de la rédaction de ce document, CPack version 3.5 prend en charge les générateurs suivants par défaut:

- Format de fichier `7Z` 7-Zip (archive)
- `IFW` Qt Installer Framework (exécutable)
- `NSIS` Null Soft Installer (exécutable)
- `NSIS64` Null Soft Installer (64 bits, exécutable)
- `STGZ` Tar GZip auto-extractible (archive)
- `TBZ2` Tar BZip2 compression (archive)
- Compression `TGZ` Tar GZip (archive)
- `TXZ` Tar XZ compression (archive)
- `TZ` Tar Compress compression (archive)
- Format de fichier `WIX` MSI via les outils WiX (archive exécutable)
- Format de fichier `ZIP` ZIP (archive)

Si aucun générateur explicite n'est fourni, CPack essaiera de déterminer le meilleur disponible en fonction de l'environnement réel. Par exemple, il préférera créer un exécutable auto-extractible sous Windows et créer uniquement une archive ZIP si aucun jeu d'outils approprié n'est trouvé.

Lire Projets d'emballage et de distribution en ligne:
https://riptutorial.com/fr/cmake/topic/4368/projets-d-emballage-et-de-distribution

Chapitre 13: Rechercher et utiliser des packages, bibliothèques et programmes installés

Syntaxe

- find_package (pkgname [version] [EXACT] [QUIET] [OBLIGATOIRE])
- inclure (FindPkgConfig)
- pkg_search_module (préfixe [OBLIGATOIRE] [QUIET] pkgname [otherpkg ...])
- pkg_check_modules (préfixe [REQUIRED] [QUIET] pkgname [otherpkg ...])

Paramètres

Paramètre	Détails
version (optionnel)	Version minimale du paquet définie par un nombre majeur et éventuellement un numéro mineur, correctif et modifié, au format major.minor.patch.tweak
EXACT (optionnel)	Indiquez que la version spécifiée dans la `version` est la version exacte à trouver
OBLIGATOIRE (facultatif)	Lève automatiquement une erreur et arrête le processus si le package est introuvable
CALME (facultatif)	La fonction n'envoie aucun message à la sortie standard

Remarques

- La méthode `find_package` est compatible sur toutes les plates-formes, tandis que la méthode `pkg-config` n'est disponible que sur les plates-formes de type Unix, telles que Linux et OSX.
- Une description complète des nombreux paramètres et options de `find_package` se trouve dans le manuel .
- Même s'il est possible de spécifier de nombreux paramètres facultatifs tels que la version du package, tous les modules de recherche n'utilisent pas tous ces paramètres correctement. Si un comportement non défini se produit, il peut être nécessaire de trouver le module dans le chemin d'installation de CMake et de corriger ou de comprendre son comportement.

Examples

Utilisez find_package et Find modules .cmake

Le moyen par défaut de rechercher des packages installés avec CMake est d'utiliser la fonction `find_package` conjointement avec un fichier `Find<package>.cmake` . Le fichier a pour but de définir les règles de recherche pour le package et de définir différentes variables, telles que `<package>_FOUND` , `<package>_INCLUDE_DIRS` et `<package>_LIBRARIES` .

De nombreux fichiers `Find<package>.cmake` sont déjà définis par défaut dans CMake. Cependant, s'il n'y a pas de fichier pour le paquet dont vous avez besoin, vous pouvez toujours écrire votre propre fichier et le mettre dans `${CMAKE_SOURCE_DIR}/cmake/modules` (ou tout autre répertoire si `CMAKE_MODULE_PATH` été remplacé)

Une liste des modules par défaut se trouve dans le manuel (v3.6) . Il est essentiel de vérifier le manuel en fonction de la version de CMake utilisée dans le projet ou il peut y avoir des modules manquants. Il est également possible de trouver les modules installés avec `cmake --help-module-list` .

Il y a un bon exemple pour un `FindSDL2.cmake` sur Github

Voici un base `CMakeLists.txt` qui nécessiterait SDL2:

```
cmake_minimum_required(2.8 FATAL_ERROR)
project("SDL2Test")

set(CMAKE_MODULE_PATH "${CMAKE_MODULE_PATH} ${CMAKE_SOURCE_DIR}/cmake/modules")
find_package(SDL2 REQUIRED)

include_directories(${SDL2_INCLUDE_DIRS})
add_executable(${PROJECT_NAME} main.c)
target_link_libraries(${PROJECT_NAME} ${SDL2_LIBRARIES})
```

Utilisez pkg_search_module et pkg_check_modules

Sur les systèmes d'exploitation de type Unix, il est possible d'utiliser le programme `pkg-config` pour rechercher et configurer des packages fournissant un fichier `<package>.pc` .

Pour pouvoir utiliser `pkg-config` , il est nécessaire d'appeler `include(FindPkgConfig)` dans un `CMakeLists.txt` . Ensuite, il y a 2 fonctions possibles:

- `pkg_search_module` , qui vérifie le package et utilise le premier disponible.
- `pkg_check_modules` , qui vérifie tous les paquets correspondants.

Voici un base `CMakeLists.txt` qui utilise `pkg-config` pour trouver SDL2 avec une version supérieure ou égale à 2.0.1:

```
cmake_minimum_required(2.8 FATAL_ERROR)
project("SDL2Test")

include(FindPkgConfig)
pkg_search_module(SDL2 REQUIRED sdl2>=2.0.1)
```

```
include_directories(${SDL2_INCLUDE_DIRS})
add_executable(${PROJECT_NAME} main.c)
target_link_libraries(${PROJECT_NAME} ${SDL2_LIBRARIES})
```

Lire Rechercher et utiliser des packages, bibliothèques et programmes installés en ligne: https://riptutorial.com/fr/cmake/topic/6752/rechercher-et-utiliser-des-packages--bibliotheques-et-programmes-installes

Chapitre 14: Test et débogage

Examples

Approche générale pour déboguer lors de la construction avec Make

Supposons que la `make` échoue:

```
$ make
```

Lancez-le à la place avec `make VERBOSE=1` pour voir les commandes exécutées. Ensuite, exécutez directement l'éditeur de liens ou la commande du compilateur que vous verrez. Essayez de le faire fonctionner en ajoutant les drapeaux ou les bibliothèques nécessaires.

Ensuite, déterminez les éléments à modifier, afin que CMake puisse transmettre des arguments corrects à la commande compiler / linker:

- quoi changer dans le système (quelles bibliothèques installer, quelles versions, quelles versions de CMake)
- si précédent échoue, quelles variables d'environnement définir ou paramètres à transmettre à CMake
- sinon, quoi changer dans `CMakeLists.txt` du projet ou les scripts de détection de la bibliothèque comme `FindSomeLib.cmake`

Pour vous aider, ajoutez des appels de `message(${MY_VARIABLE})` dans `CMakeLists.txt` ou `*.cmake` pour déboguer les variables que vous souhaitez inspecter.

Laisser CMake créer des Makefiles verbeux

Une fois qu'un projet CMake est initialisé via `project()` , la verbosité de sortie du script de génération résultant peut être ajustée via:

```
CMAKE_VERBOSE_MAKEFILE
```

Cette variable peut être définie via la ligne de commande de CMake lors de la configuration d'un projet:

```
cmake -DCMAKE_VERBOSE_MAKEFILE=ON <PATH_TO_PROJECT_ROOT>
```

Pour GNU, cette variable a le même effet que `make VERBOSE=1` .

Déboguer les erreurs find_package ()

Remarque: Les messages d'erreur CMake affichés incluent déjà le correctif pour les chemins d'installation de bibliothèque / outil "non standard". Les exemples suivants illustrent simplement

des sorties CMake `find_package()` plus détaillées.

CMake pris en charge en interne Package / Module

Si le code suivant (remplace le module `FindBoost` par votre module en question)

```
cmake_minimum_required(VERSION 2.8)
project(FindPackageTest)

find_package(Boost REQUIRED)
```

donne une erreur comme

```
CMake Error at [...]/Modules/FindBoost.cmake:1753 (message):
  Unable to find the requested Boost libraries.

  Unable to find the Boost header files.  Please set BOOST_ROOT to the root
  directory containing Boost or BOOST_INCLUDEDIR to the directory containing
  Boost's headers.
```

Et vous vous demandez où il a essayé de trouver la bibliothèque, vous pouvez vérifier si votre paquet a une option `_DEBUG` comme le module `Boost` a pour obtenir une sortie plus détaillée

```
$ cmake -D Boost_DEBUG=ON ..
```

CMake activé Package / Bibliothèque

Si le code suivant (remplace le `Xyz` avec votre bibliothèque en question)

```
cmake_minimum_required(VERSION 2.8)
project(FindPackageTest)

find_package(Xyz REQUIRED)
```

donne une erreur comme

```
CMake Error at CMakeLists.txt:4 (find_package):
  By not providing "FindXyz.cmake" in CMAKE_MODULE_PATH this project has
  asked CMake to find a package configuration file provided by "Xyz", but
  CMake did not find one.

  Could not find a package configuration file provided by "Xyz" with any of
  the following names:

    XyzConfig.cmake
    xyz-config.cmake
```

```
  Add the installation prefix of "Xyz" to CMAKE_PREFIX_PATH or set "Xyz_DIR"
  to a directory containing one of the above files.  If "Xyz" provides a
  separate development package or SDK, be sure it has been installed.
```

Et vous vous demandez où il a essayé de trouver la bibliothèque, vous pouvez utiliser la variable globale `CMAKE_FIND_DEBUG_MODE` non `CMAKE_FIND_DEBUG_MODE` pour obtenir une sortie plus détaillée

```
$ cmake -D CMAKE_FIND_DEBUG_MODE=ON ..
```

Lire Test et débogage en ligne: https://riptutorial.com/fr/cmake/topic/4098/test-et-debogage

Chapitre 15: Utilisation de CMake pour configurer les tags préprocesseurs

Introduction

L'utilisation de CMake dans un projet C ++ s'il est utilisé correctement peut permettre au programmeur de se concentrer moins sur la plate-forme, le numéro de version du programme et plus encore sur le programme lui-même. Avec CMake, vous pouvez définir des balises de préprocesseur qui permettent de vérifier facilement la plate-forme ou les autres balises de préprocesseur dont vous avez besoin dans le programme. Tels que le numéro de version qui pourrait être utilisé dans un système de journalisation.

Syntaxe

- #define nom_processeur "@ cmake_value @"

Remarques

Il est important de comprendre que tous les préprocesseurs ne doivent pas être définis dans le `config.h.in` . Les balises de préprocesseur sont généralement utilisées uniquement pour faciliter la vie des programmeurs et doivent être utilisées avec discrétion. Vous devez rechercher si une balise de préprocesseur existe déjà avant de la définir car vous risquez de rencontrer un comportement indéfini sur un système différent.

Examples

Utiliser CMake pour définir le numéro de version pour une utilisation C ++

Les possibilités sont infinies. comme vous pouvez utiliser ce concept pour extraire le numéro de version de votre système de génération; comme git et utilisez ce numéro de version dans votre projet.

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.8)
project(project_name VERSION "0.0.0")

configure_file(${path to configure file 'config.h.in'}
include_directories(${PROJECT_BINARY_BIN}) // this allows the 'config.h' file to be used
throughout the program

...
```

config.h.in

```
#ifndef INCLUDE_GUARD
#define INCLUDE_GUARD

#define PROJECT_NAME "@PROJECT_NAME@"
#define PROJECT_VER  "@PROJECT_VERSION@"
#define PROJECT_VER_MAJOR "@PROJECT_VERSION_MAJOR@"
#define PROJECT_VER_MINOR "@PROJECT_VERSION_MINOR@"
#define PTOJECT_VER_PATCH "@PROJECT_VERSION_PATCH@"

#endif // INCLUDE_GUARD
```

main.cpp

```
#include <iostream>
#include "config.h"
int main()
{
    std::cout << "project name: " << PROJECT_NAME << " version: " << PROJECT_VER << std::endl;
    return 0;
}
```

sortie

```
project name: project_name version: 0.0.0
```

Lire Utilisation de CMake pour configurer les tags préprocesseurs en ligne: https://riptutorial.com/fr/cmake/topic/10885/utilisation-de-cmake-pour-configurer-les-tags-preprocesseurs

Chapitre 16: Variables et propriétés

Introduction

La simplicité des variables CMake de base contredit la complexité de la syntaxe de la variable complète. Cette page documente les différents cas variables, avec des exemples, et indique les pièges à éviter.

Syntaxe

- set (variable_name value [description du type CACHE [FORCE]])

Remarques

Les noms de variables sont sensibles à la casse. Leurs valeurs sont de type string. La valeur d'une variable est référencée via:

```
${variable_name}
```

et est évalué dans un argument cité

```
"${variable_name}/directory"
```

Examples

Variable en cache (globale)

```
set(my_global_string "a string value"
    CACHE STRING "a description about the string variable")
set(my_global_bool TRUE
    CACHE BOOL "a description on the boolean cache entry")
```

Si une variable en cache est déjà définie dans le cache lorsque CMake traite la ligne correspondante (par exemple, lorsque CMake est réexécutée), elle n'est pas modifiée. Pour remplacer la valeur par défaut, ajoutez `FORCE` comme dernier argument:

```
set(my_global_overwritten_string "foo"
    CACHE STRING "this is overwritten each time CMake is run" FORCE)
```

Variable locale

```
set(my_variable "the value is a string")
```

Par défaut, une variable locale est uniquement définie dans le répertoire en cours et tous les sous-répertoires ajoutés via la commande `add_subdirectory` .

Pour étendre la portée d'une variable, il y a deux possibilités:

1. `CACHE` le rendra disponible à l'échelle mondiale
2. utilisez `PARENT_SCOPE` , ce qui le rendra disponible dans la portée parent. La portée parent est soit le fichier `CMakeLists.txt` dans le répertoire parent, soit l'appelant de la fonction en cours.

   Techniquement, le répertoire parent sera le fichier `CMakeLists.txt` qui incluait le fichier actuel via la commande `add_subdirectory` .

Cordes et Listes

Il est important de savoir comment CMake distingue les listes et les chaînes simples. Lorsque vous écrivez:

```
set(VAR "ab c")
```

vous créez une **chaîne** avec la valeur `"ab c"` . Mais quand vous écrivez cette ligne sans guillemets:

```
set(VAR abc)
```

Vous créez une **liste** de trois éléments à la place: `"a"` , `"b"` et `"c"` .

Les variables non listées sont aussi des listes (d'un seul élément).

Les listes peuvent être utilisées avec la commande `list()` , qui permet de concaténer des listes, de les rechercher, d'accéder à des éléments arbitraires, etc. (documentation de list ()).

Un peu déroutant, une **liste** est aussi une **chaîne** . La ligne

```
set(VAR abc)
```

est équivalent à

```
set(VAR "a;b;c")
```

Par conséquent, pour concaténer des listes, on peut aussi utiliser la commande `set()` :

```
set(NEW_LIST "${OLD_LIST1};${OLD_LIST2})"
```

Variables et le cache des variables globales

Vous utiliserez principalement des "variables normales" :

```
set(VAR TRUE)
set(VAR "main.cpp")
set(VAR1 ${VAR2})
```

Mais CMake connaît également les "variables en cache" globales (persistantes dans `CMakeCache.txt`). Et si des variables normales et en cache du même nom existent dans la portée actuelle, les variables normales masquent celles en cache:

```
cmake_minimum_required(VERSION 2.4)
project(VariablesTest)

set(VAR "CACHED-init" CACHE STRING "A test")
message("VAR = ${VAR}")

set(VAR "NORMAL")
message("VAR = ${VAR}")

set(VAR "CACHED" CACHE STRING "A test" FORCE)
message("VAR = ${VAR}")
```

Première sortie

```
VAR = CACHED-init
VAR = NORMAL
VAR = CACHED
```

Sortie de la deuxième course

```
VAR = CACHED
VAR = NORMAL
VAR = CACHED
```

Remarque: L'option `FORCE` désactive / supprime également la variable normale de la portée actuelle.

Cas d'utilisation pour les variables mises en cache

Il y a généralement deux cas d'utilisation (veuillez ne pas les utiliser pour les variables globales):

1. Une valeur dans votre code doit être modifiable par l'utilisateur de votre projet, par exemple avec l' `cmakegui` , `ccmake` ou `cmake -D ...` :

   CMakeLists.txt / MyToolchain.cmake

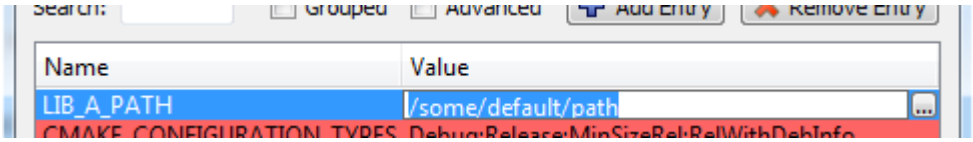
   ```
   set(LIB_A_PATH "/some/default/path" CACHE PATH "Path to lib A")
   ```

   Ligne de commande

   ```
   $ cmake -D LIB_A_PATH:PATH="/some/other/path" ..
   ```

   Cela prédéfinit cette valeur dans le cache et la ligne ci-dessus ne le modifiera pas.

CMake GUI



Dans l'interface graphique, l'utilisateur lance d'abord le processus de configuration, puis modifie toute valeur mise en cache et termine le démarrage de la génération de l'environnement de génération.

2. De plus, CMake met en cache les résultats de recherche / test / identification du compilateur (il n'a donc pas besoin de le refaire à chaque fois qu'il réexécute les étapes de configuration / génération)

```
find_path(LIB_A_PATH libA.a PATHS "/some/default/path")
```

Ici, `LIB_A_PATH` est créé en tant que variable mise en cache.

Ajout d'indicateurs de profilage à CMake pour utiliser gprof

La série d'événements est censée fonctionner comme suit:

1. Compiler le code avec l'option -pg
2. Code de liaison avec l'option -pg
3. Exécuter de programme
4. Le programme génère le fichier gmon.out
5. Exécuter le programme gprof

Pour ajouter des indicateurs de profilage, vous devez ajouter à votre CMakeLists.txt:

```
SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -pg")
SET(CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} -pg")
SET(CMAKE_SHARED_LINKER_FLAGS "${CMAKE_SHARED_LINKER_FLAGS} -pg")
```

Cela doit ajouter des drapeaux pour compiler et lier, et utiliser après exécution du programme:

gprof ./my_exe

Si vous obtenez une erreur comme:

```
gmon.out: No such file or directory
```

Cela signifie que la compilation n'a pas ajouté d'informations de profilage correctement.

Lire Variables et propriétés en ligne: https://riptutorial.com/fr/cmake/topic/2091/variables-et-proprietes

Crédits

S. No	Chapitres	Contributeurs
1	Commencer avec cmake	Amani, arrowd, ComicSansMS, Community, Daniel Schepler, dontloo, Fantastic Mr Fox, fedepad, Florian, greatwolf, Mario, Neui, OliPro007, Torbjörn, Ziv
2	Ajouter des répertoires au chemin d'inclusion du compilateur	kiki, Torbjörn
3	Configurations de construction	Jav_Rock
4	Configurer le fichier	Jav_Rock, Shihe Zhang, vgonisanz
5	Construire des cibles	arrowd, Neui, Torbjörn
6	Créer des suites de test avec CTest	arrowd, ComicSansMS, Torbjörn
7	Étapes de construction personnalisées	Developer Paul
8	Fonctions de compilation et sélection standard C / C ++	wasthishelpful
9	Fonctions et macros	Torbjörn
10	Intégration CMake dans les outils GitHub CI	ComicSansMS
11	Projet hiérarchique	Adam Trhon, Anedar, Clare Macrae, Robert
12	Projets d'emballage et de distribution	Mario, Meysam, Neui
13	Rechercher et utiliser des packages, bibliothèques et	OliPro007

	programmes installés	
14	Test et débogage	Florian, Torbjörn, Velkan
15	Utilisation de CMake pour configurer les tags préprocesseurs	JVApen, Matthew
16	Variables et propriétés	arrowd, CivFan, Florian, Torbjörn, Trilarion, Trygve Laugstøl, vgonisanz