CMAKE le système de construction logicielle

Table of Contents

CMake est un système de construction logicielle multiplateforme. Il permet de vérifier les prérequis nécessaires à la construction, de déterminer les dépendances entre les différents composants d'un projet, afin de planifier une construction ordonnée et adaptée à la plateforme. La construction du projet est ensuite déléguée à un logiciel spécialisé dans l'ordonnancement de taches et spécifique à la plateforme, Make, Ninja ou Microsoft Visual Studio.

RPATH

Lorsqu'on lie une bibliothèque partagée (*.so sous Linux), l'exécutable doit savoir d'une manière ou d'une autre où rechercher ladite bibliothèque lors de l'exécution. Dans la plupart des cas, la bibliothèque serait placée dans un chemin de bibliothèque système commun et l'exécutable la trouverait grâce à une liste prédéfinie d'endroits à rechercher. Toutefois, si on souhaite expédier le fichier de bibliothèque partagée avec un exécutable ou si plusieurs versions sont installées en parallèle, il faut s'assurer que cmake trouve la bibliothèque à son emplacement personnalisé.

Avec **RPATH** (**R**un-time Search **Path**), on peut résoudre ce problème en incorporant des chemins de recherche supplémentaires directement dans l'exécutable.

Deux points généraux doivent être considérés avec **CMAKE**:
* Pour éviter les anciens comportements et avoir à gérer les politiques CMake, il faut utiliser CMake version>3.13.
* **RPATH** peut également être utile pendant le développement, car on peut lier des bibliothèques dans l'arborescence de construction par rapport à l'exécutable.

Relativité

Dans de nombreux cas et pour une portabilité maximale, on ne doit pas spécifier un chemin absolu vers la bibliothèque partagée, mais l'avoir par rapport à l'exécutable. Pour cela, on a **$ORIGIN**.

Sous Linux, **$ORIGIN** représente au moment de l'exécution l'emplacement de l'exécutable et, par conséquent, si on définit le RPATH sur, par exemple, `$ORIGIN/lib/libmylib.so`, l'exécutable recherchera la bibliothèque partagée nécessaire par rapport à l'emplacement de l'exécutable.

Les Variables

CMake propose de nombreuses options pour affiner le comportement lors de la liaison de l'arborescence de construction et de la liaison d'installation. On n'entrera pas dans les détails de la liste suivante de variables CMake liées à RPATH. Pour en savoir plus, consulter la documentation liée:

- **CMAKE_INSTALL_RPATH** - Une liste séparée par des points-virgules spécifiant le RPATH à utiliser dans les cibles installées
- **CMAKE_SKIP_RPATH** - Si true, n'ajoute pas d'informations sur le chemin d'exécution
- **CMAKE_SKIP_BUILD_RPATH** - Ne pas inclure les RPATH dans l'arborescence de construction
- **CMAKE_BUILD_WITH_INSTALL_RPATH** - Utilise le chemin d'installation pour le RPATH
- **CMAKE_INSTALL_RPATH_USE_LINK_PATH** - Ajouter des chemins à la recherche de l'éditeur de liens et au RPATH installé

Comportement par défaut

Par défaut, **RPATH** sera utilisé dans l'arborescence de construction, mais effacé lors de l'installation des cibles, laissant un **RPATH** vide.

Avec les paramètres suivants, on peut s'assurer que les bibliothèques seront également trouvées dans l'arborescence de construction lorsqu'on installera les cibles.

```
# ne pas sauter le RPATH complet pour l'arborescence de construction
set(CMAKE_SKIP_BUILD_RPATH FALSE)

# lors de la construction, ne pas utiliser déjà le RPATH d'installation
# (mais plus tard lors de l'installation)
set(CMAKE_BUILD_WITH_INSTALL_RPATH FALSE)
set(CMAKE_INSTALL_RPATH "${CMAKE_INSTALL_PREFIX}/lib")

# ajouter les parties déterminées automatiquement du RPATH
# qui pointent vers des répertoires en dehors de l'arborescence
# de construction vers le RPATH d'installation
set(CMAKE_INSTALL_RPATH_USE_LINK_PATH TRUE)
```

Pour ignorer complètement **RPATH** on peut définir `CMAKE_SKIP_RPATH`.

Chemin spécifique

Lorsqu'on souhaite compiler un programme avec **CMAKE** et l'installer un logiciel localement dans un chemin spécifique (par exemple `$HOME/.local/`) au lieu d'un dossier `/usr/`.

Après l'installation, les fichiers binaires et les bibliothèques du logiciel sont stockés respectivement dans `$HOME/.local/bin/` et `$HOME/.local/lib/`. Cependant, lorsqu'pn essaie d'exécuter le programme, il génère une erreur indiquant que la bibliothèque requise n'est pas trouvée (qui, soit dit cependant, est présente dans `$HOME/.local/lib/`).

Le programme fonctionne correctement si on règle `$LD_LIBRARY_PATH` sur `$HOME/.local/lib`. Mais au lieu de cela, on peut spécifier la variable **RPATH** (qui pointerait vers `$HOME/.local/lib`) lors de la compilation du logiciel à l'aide de **CMAKE**, en utilisant la ligne suivante dans le CMakefile:

```
set(CMAKE_INSTALL_RPATH "${CMAKE_INSTALL_PREFIX}/lib")
```

CMAKE_INSTALL_RPATH est une liste prédéfinie, donc on peut préférer utiliser une liste (`APPEND CMAKE_INSTALL_RPATH ${CMAKE_INSTALL_PREFIX}/lib`). Si on inclue (`GNUInstallDirs`), on peut également utiliser (`APPEND CMAKE_INSTALL_RPATH ${CMAKE_INSTALL_LIBDIR}`)

On peut également utiliser :

```
set(CMAKE_BUILD_RPATH "/my/libs/location")
```

spécifiant les entrées de chemin d'exécution (RPATH) à ajouter aux fichiers binaires liés dans l'arborescence de construction (pour les plates-formes qui le prennent en charge). Les entrées ne seront pas utilisées pour les fichiers binaires dans l'arborescence d'installation.

Les Bibliothèques partagées

Les liens dynamiques

Lorsque l'exécutable est généré, l'option **-lxxx** spécifie le fichier de bibliothèque à lier. Par exemple si on spécifie une bibliothèque qui doit être liée, le connecteur enregistrera les informations de la bibliothèque dans l'en-tête du fichier exécutable. Ensuite, lorsque l'exécutable est en cours d'exécution, le chargeur dynamique lit les informations d'en-tête et charge toutes les bibliothèques de liens. Dans le processus, si l'utilisateur spécifie un lien vers une bibliothèque non liée, la bibliothèque sera également chargée lors de l'exécution de l'exécutable final. S'il existe de nombreuses bibliothèques non liées comme celle-ci, cela ralentira évidemment le processus de démarrage du programme.

GCC/G++ fournit deux options **–as-needed** et **–no-as-needed**:

- **–as-needed**, permet à l'éditeur de liens d'ignorer, c'est-à-dire de ne pas lier certaines des bibliothèques fournies sur sa ligne de commande si elles ne sont pas réellement utilisées par la bibliothèque partagée en cours de création. Par exemple, si on indique l'option -lm sur la ligne de commande à l'éditeur de liens mais qu'on n'utilise aucune fonction mathématique, la bibliothèque `libm.so` ne sera pas liée.
- **–no-as-needed** force à ne pas effectuer une telle vérification.

C'est une excellente idée d'introduire cette option car elle oblige les développeurs à réfléchir plus attentivement à leurs dépendances de bibliothèque. Mais l'introduction de cette option par défaut peut cependant casser pas mal de builds.

Afin de corriger les défauts inhérents aux builds que cette option expose. Il existe deux options supplémentaires de l'éditeur de liens qui peuvent énormément aider:

- **–no-undefined**, vérifie que la bibliothèque partagée en cours de compilation ne se retrouvera pas avec des symboles qui ne sont résolus par aucun par ses dépendances
- **–[no-]allow-shlib-undefined**, vérifie qu'aucune des bibliothèques partagées dépendantes fournies à l'éditeur de liens n'a elle-même de symboles non résolus

Pour utiliser ces options depuis la ligne de commande gcc, il faut utiliser le préfixe **-Wl**. Ainsi, par exemple, `-Wl,--no-allow-shlib-undefined`.

Lors de la liaison d'un binaire, on peut utiliser **CMAKE_EXE_LINKER_FLAGS** pour ajouter un indicateur (par ex. **-Wl,-as-needed**). Cependant, si on lie une bibliothèque, ce drapeau supplémentaire ne sera pas pris en compte. **CMake** a une meilleure solution, en utilisant **CMAKE_SHARED_LINKER_FLAGS** comme ceci:

```
set (CMAKE_SHARED_LINKER_FLAGS "-Wl,--as-needed")
```

L'option **BR2_TARGET_LDFLAGS** dans **Buildroot**, permet d'ajouter ces flags personalisés (`TARGET_LDFLAGS+=-L$(STAGING_DIR)/lib -L$(STAGING_DIR)/usr/lib $(BR2_TARGET_LDFLAGS)`):

`BR2_TARGET_LDFLAGS="-Wl,-dynamic-linker,/lib/ld-linux.so.2 -Wl,--as-needed"`

Symboles indéfinis

Par défaut, l'éditeur de liens GNU interdit les références de symboles indéfinis lors de la liaison des exécutables finaux, mais les autorise pour les bibliothèques partagées. Ce dernier comportement est nécessaire car de nombreux projets dépendent des symboles de bibliothèque partagée manquants fournis par les exécutables auxquels ils sont liés ou chargés. C'est généralement le cas avec les plugins, qui appellent l'API du plugin fournie par l'exécutable qui les charge.

Cependant, ce comportement signifie également que si une bibliothèque partagée n'est pas liée à une dépendance, elle se construira correctement et l'erreur ne sera pas visible tant que le premier exécutable lié à la bibliothèque (directement ou indirectement) ne sera pas construit. Par exemple, si **app-accessibility/flite** a des symboles manquants et des liens **media-video/ffmpeg** vers celui-ci, aucun de ces deux packages n'échouera à se construire. Cependant, un paquet utilisant **ffmpeg**, par ex. **media-video/mpv** ne pourra pas être construit en raison de symboles indéfinis dans **flite**.

- **-Wl,-z,defs** (ou son équivalent **–no-undefined**) oblige l'éditeur de liens à traiter tous les symboles indéfinis dans les fichiers objets liés (y compris les bibliothèques statiques) comme des erreurs. Avec cette option, tous les symboles référencés dans le code inclus dans la bibliothèque partagée doivent être résolus à l'aide de bibliothèques supplémentaires liées. En d'autres termes, cette option empêche l'éditeur de liens de créer des bibliothèques partagées auxquelles il manque des dépendances directes.
- **-Wl,–no-allow-shlib-undefined** oblige l'éditeur de liens à traiter tous les symboles non définis dans les bibliothèques partagées liées comme des erreurs. En d'autres termes, cette option empêche l'éditeur de liens de créer des bibliothèques partagées auxquelles il manque des dépendances indirectes.

Par exemple, supposons que **app-accessibility/flite** ait des symboles manquants dans la bibliothèque. **-Wl,-z,defs** sur ce paquet entraînerait l'échec de sa construction en raison de symboles non résolus. Si la bibliothèque est déjà construite, **media-video/ffmpeg** se construira avec succès puisque les symboles non résolus se trouvent dans une bibliothèque partagée existante. Si **-Wl,–no-allow-shlib-undefined** était utilisé, la construction de **ffmpeg** échouerait.

Utilsation d'un éditeur de liens personnalisé

Afin de pouvoir spécifier un éditeur de liens (ld) personnalisé, une option documentée dans **gcc** est d'utiliser **-fuse-ld** :

```
cmake -DCMAKE_CXX_FLAGS="-fuse-ld=lld"
```

g++ ou **clang++** recevront l'indicateur **-fuse-ld=lld** à chaque appel, et lorsqu'ils feront une liaison, ils utiliseront la commande spécifiée au lieu de la valeur par défaut intégrée.

L'option est analysée ainsi `(-f) (use-ld) (=) (lld)`, et lorsqu'on utilise **Clang**, **lld** peut être remplacé par n'importe quelle autre commande de l'éditeur de liens que l'on souhaite utiliser, comme ld.exe, ld.gold, mingw32/bin/ld.exe, etc.

GCC n'est pas aussi flexible, son **-fuse-ld** n'accepte qu'un des trois arguments possibles : **lld**, **bfd** ou **gold**. Il invoquera le premier exécutable correspondant qu'il trouve sur le PATH.

L'utilisation de **CMAKE_C_LINKER_EXECUTABLE** et **CMAKE_CXX_LINKER_EXECUTABLE** n'est pas

la bonne approche, car il faut spécifier la ligne de lien entière, pas juste le lien.

L'utilisation de **-DCMAKE_LINKER=/path/to/my/linker** est la bonne façon de faire cela, mais pour un exécutable **C++** ce qui suit est utilisé pour créer la ligne de commande de lien dans `Modules/CMakeCXXInformation.cmake`:

```
    if(NOT CMAKE_CXX_LINK_EXECUTABLE)
      set(CMAKE_CXX_LINK_EXECUTABLE
         "<CMAKE_CXX_COMPILER>  <FLAGS> <CMAKE_CXX_LINK_FLAGS>
<LINK_FLAGS> <OBJECTS>  -o <TARGET> <LINK_LIBRARIES>")
    endif()
```

Comme on peut le voir, la ligne de commande du lien définie dans `Modules/CMake{C,CXX,Fortran}Information.cmake` utilise par défaut le compilateur, et non **CMAKE_LINKER** (le compilateur C++ **<CMAKE_CXX_COMPILER>** est utilisé par défaut comme frontal pour lien exécutables C++). Cela peut être changé en remplaçant la règle qui construit la ligne de commande du lien, qui réside dans les variables **CMAKE_CXX_LINK_EXECUTABLE** (cette variable n'indique pas le chemin vers l'exécutable de l'éditeur de liens ; elle dit comment lier un exécutable):

Afin d'utiliser l'éditeur de liens que l'on souhaite, il faut donc définir un **CMAKE_CXX_LINK_EXECUTABLE** qui utilise **“\<CMAKE_LINKER>“** dans sa définition de la ligne de commande de l'éditeur de liens.

Une approche consiste à définir cette règle pour utiliser l'éditeur de liens, par exemple:

```
cmake -DCMAKE_LINKER=/path/to/linker -
DCMAKE_CXX_LINK_EXECUTABLE="<CMAKE_LINKER> <FLAGS> <CMAKE_CXX_LINK_FLAGS>
<LINK_FLAGS> <OBJECTS> -o <TARGET> <LINK_LIBRARIES>"
```

Cross compilatiop avec cmake

La cross-compilation permet de générer sur une architecture X un exécutable qui sera compatible avec une architecture Y. La bonne façon de cross-compiler avec CMake est d'avoir un fichier de chaîne d'outils :

Exemple toolchain.cmake :

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_C_COMPILER /opt/foo-toolchain/usr/bin/mipsel-foo-linux-uclibc-gcc)
set(CMAKE_CXX_COMPILER /opt/foo-toolchain/usr/bin/mipsel-foo-linux-uclibc-
g++)
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -D_LARGEFILE_SOURCE -
D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64  -pipe -O2 " CACHE STRING
"Buildroot CFLAGS" FORCE)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -D_LARGEFILE_SOURCE -
D_LARGEFILE64_SOURCE -D_FILE_OFFSET_BITS=64  -pipe -O2 " CACHE STRING
"Buildroot CXXFLAGS" FORCE)
```

```
set(CMAKE_INSTALL_SO_NO_EXE 0)
set(CMAKE_PROGRAM_PATH "/opt/foo-toolchain/usr/bin")
set(CMAKE_FIND_ROOT_PATH "/opt/foo-toolchain/usr/mipsel-foo-linux-
uclibc/sysroot")
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(ENV{PKG_CONFIG_SYSROOT_DIR} "/opt/foo-toolchain/usr/mipsel-foo-linux-
uclibc/sysroot")
```

Il faut s'assurer de définir les variables **CMAKE_SYSROOT** et **CMAKE_STAGING_PREFIX** ainsi

```
set(CMAKE_SYSROOT /home/devel/rasp-pi-rootfs)
set(CMAKE_STAGING_PREFIX /home/devel/stage)
```

Ensuite, on peut effectuer une compilation croisée avec le fichier CMakeLists.txt non modifié en faisant :

```
cmake -DCMAKE_TOOLCHAIN_FILE=toolchain.cmake ..
```

Buildroot fournit un fichier `toolchainfile.cmake` pour **CMAKE** permettant :
- de relocaliser la chaîne d'outils, en se basant sur l'emplacement du fichier `$(HOST_DIR)/share/buildroot`
- de faire pointer cmake vers l'emplacement où se trouve le modules personnalisés **CMAKE_CURRENT_LIST_DIR**
- de définir les {C,CXX}FLAGS ajoutés par CMake en fonction du type de construction défini par Buildroot.

Ce fichier de chaîne d'outils peut être utilisé à la fois à l'intérieur et à l'extérieur de **Buildroot**.

Si on utilise le fichier `toolchainfile.cmake` en dehors de **Buildroot** et qu'on personnalise les drapeaux du compilateur/éditeur de liens, alors il faut:
- les définir tous sur la ligne de commande cmake, par exemple : `cmake -DCMAKE_C_FLAGS="@@TARGET_CFLAGS@@ -Dsome_custom_flag" ...`
- s'assurer que le code CMake du projet les étend comme ceci si nécessaire : `set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Dsome_definitions")`