INTRODUCTION À CMAKE

Laurent NOËL

25/02/2015

Université Paris-Est

LES BASES

QU’EST CE QUE CMAKE ?

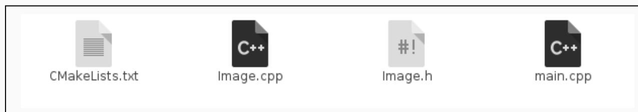
CMake est un programme multiplate-forme permettant de produire une solution de compilation pour un projet. Il pourra par exemple génerer un ensemble de Makefile, un projet Visual Studio, un projet Code::Blocks, etc. Le projet peut ensuite être compilé avec la solution choisie.

LE FICHIER CMAKELISTS.TXT

Le fichier *CMakeLists.txt* doit être écrit par l'utilisateur et fourni en entrée à CMake.

La syntaxe est très simple: le fichier est constitué d'une suite de commandes décrivant le processus de compilation du projet.

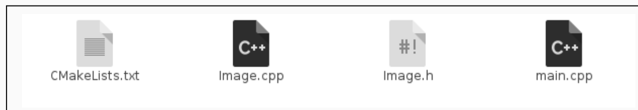
```
nom_commande(argument1 argument2 ...)
```



Listing de tous les fichiers:

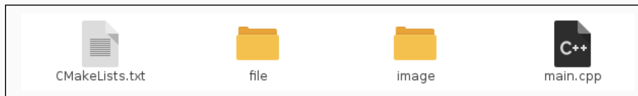
```
cmake_minimum_required(VERSION 2.8)
project(ProjetSimpleCMake)
set(SRC_FILES main.cpp
      Image.cpp
      Image.h)
add_executable(ProjetSimple1 ${SRC_FILES})
```

EXEMPLE - PROJET SIMPLE



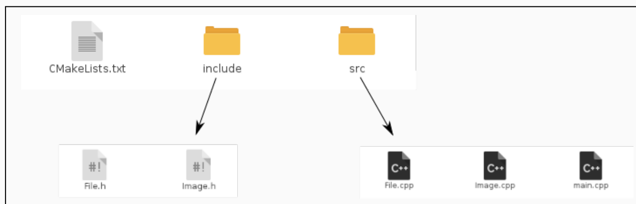
Calcul de la liste des fichiers à partir d'un pattern:

```
cmake_minimum_required(VERSION 2.8)
project(ProjetSimpleCMake)
file(GLOB SRC_FILES *.cpp *.h)
add_executable(ProjetSimple1 ${SRC_FILES})
```



Calcul récursif de la liste des fichiers à partir d'un pattern:

```
cmake_minimum_required(VERSION 2.8)
project(ProjetSimpleCMake)
file(GLOB_RECURSE SRC_FILES *.cpp *.h)
add_executable(ProjetSimple1 ${SRC_FILES})
```



On utilise la commande include_directories:

```
cmake_minimum_required(VERSION 2.8)
project(ProjetSimpleCMake)
include_directories(include)
file(GLOB_RECURSE SRC_FILES *.cpp *.h)
add_executable(ProjetSimple1 ${SRC_FILES})
```

Cette commande peut prendre plusieurs repertoires en argument. Il est possible de l'appeler plusieurs fois pour ajouter itérativement des répertoires.

Bonne pratique: séparer completement le repertoire de build du repertoire source contenant le CMakeLists.txt.

- mon-projet
  - mon-projet-build
    - CMakeCache.txt
    - ...
  - mon-projet
    - CMakeLists.txt
    - ...

Compilation du projet:

```
cd mon-projet/mon-projet-build
cmake ../mon-projet
make
```

Après un premier lancement de CMake dans le repertoire build, il n'est plus necessaire de préciser le chemin vers le repertoire source pour relancer CMake:

```
cd mon-projet/mon-projet-build
cmake .
make
```

Interet:

- Plus facile d'exclure le repertoire build d'un dépot (GIT par exemple)
- Conserve le repertoire source ”propre” (pas de fichiers temporaires dans son arborescence)
- Les commandes CMake ne risquent pas d'entrer en conflit les fichiers du repertoire build (commandes récursives par exemple)

BIBLIOTHÈQUES

PROJETS AVEC BIBLIOTHÈQUES INSTALLÉES SUR LE SYSTÈME

Utilisation de la SDL:

```
find_package(SDL REQUIRED)
[...]
include_directories(${SDL_INCLUDE_DIR})
[...]
add_executable(MonExecutable ${SRC_FILE})
target_link_libraries(MonExecutable ${SDL_LIBRARY})
```

Attention

Le nom des variables définies par les bibliothèques n'est pas standard.

Utilisation d'OpenGL:

```
find_package(OpenGL REQUIRED)
[...]
include_directories(${OPENGL_INCLUDE_DIR})
[...]
add_executable(MonExecutable ${SRC_FILE})
target_link_libraries(MonExecutable ${OPENGL_LIBRARIES})
```

Attention

La SDL définit la variable SDL_LIBRARY alors qu'OpenGL définit OPENGL_LIBRARIES.

PROJETS AVEC BIBLIOTHÈQUES INSTALLÉES SUR LE SYSTÈME

La commande find_package(Lib) recherche un fichier FindLib.cmake dans le repertoire d'installation de CMake.
Le repertoire par default sous Linux est /usr/share/cmake-3.1/Modules/ (variable selon la version de CMake installée). Ce fichier spécifie généralement les variables qu'il définit dans un commentaire au début du fichier.

PROJETS AVEC BIBLIOTHÈQUES INSTALLÉES SUR LE SYSTÈME

Si le fichier FindLib.cmake n'existe pas, il est toujours possible soit de l'écrire (compliqué), soit d'en trouver un sur le net.
On le place alors dans un sous répertoire du projet (qu'on appelle "CMake" généralement) et on insère la ligne suivante dans le CMakeLists.txt avant de faire les appels à find_package:

```
set(CMAKE_MODULE_PATH CMake)
```

Arborescence:

- mon-projet
    - CMakeLists.txt
    - CMake
        - FindBidule.cmake
        - FindRORPO.cmake
    - ...

COMPILER UNE BIBLIOTHÈQUE AVEC CMAKE

Il suffit d'utiliser la commande add_library à la place de add_executable:

```
cmake_minimum_required(VERSION 2.8)
project(libRORPO)
file(GLOB_RECURSE SRC_FILES *.cpp *.h)
add_library(RORPO ${SRC_FILES})
```

PROJETS AVEC BIBLIOTHÈQUES INTERNE À COMPILER

Lorsque la bibliothèque est sous la forme d'un projet CMake, il est possible de l'intégrer en tant que sous projet.

Arborescence:

- mon-projet
  - CMakeLists.txt
  - main.cpp
  - libRORPO
    - CMakeLists.txt
    - ...

Dans le CMakeLists.txt de plus haut niveau, on ajoute la ligne:

```
add_subdirectory(libRORPO)
```

On link ensuite notre executable à la bibliothèque:

```
add_executable(MonExecutable ${SRC_FILES})
target_link_libraries(MonExecutable RORPO)
```

ALLER PLUS LOIN

MODIFIER L'ORGANISATION DU REPERTOIRE BUILD

Il peut être utile de modifier l'organisation du repertoire de build, en particulier pour placer les executables et les bibliothèques dans des repertoires particuliers séparés.

Pour cela il suffit de modifier les variables CMake suivantes:

```
# Executables compilés dans le repertoire build/bin:
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)
# Bibliothèques compilés dans le repertoire build/lib:
set(LIBRARY_OUTPUT_PATH ${PROJECT_BINARY_DIR}/lib)
```

La variable PROJECT_BINARY_DIR est définie par CMake et contient le chemin du repertoire de build.

On veut parfois copier des fichiers assets (images, modèles 3D, fichiers de configuration, shaders, scripts, ...) dans le repertoire de build à coté des executables afin d'avoir un chemin relatif à l'executable pour les charger.
Le code C++ pour charger un fichier situé à coté de l'executable peut alors s'écrire de la manière suivante:

```
int main(int argc, char** argv) {
    std::string pathToExe = argv[0];
    std::string exeDir = "";
    size_t slashPosition = pathToExe.find_last_of('/');
    if(slashPosition != std::string::npos) {
        exeDir = pathToExe.substr(0, slashPosition);
    }
    std::string gnuPlotFilePath = exeDir + "/plotMyAss.plot";
    std::cerr << gnuPlotFilePath << std::endl;
    return 0;
}
```

GESTION DE FICHIERS ASSETS

Une façon simple est d'utiliser la commande file de CMake:

```
file(COPY plotMyAss.plot
     DESTINATION ${EXECUTABLE_OUTPUT_PATH}/plotMyAss.plot)
```

Le problème de cette solution est qu'il faut relancer CMake lorsque l'on modifie les fichiers à copier. Si on oublie, l'executable utilisera une ancienne version des fichiers, ce qui n'est pas souhaitable.

La solution compliquée mais propre est de créer pour chaque fichier à copier une commande personnalisée:

```
add_custom_command(
    OUTPUT ${EXECUTABLE_OUTPUT_PATH}/plotMyAss.plot
    COMMAND
        ${CMAKE_COMMAND} -E copy
        ${CMAKE_CURRENT_SOURCE_DIR}/plotMyAss.plot
        ${EXECUTABLE_OUTPUT_PATH}/plotMyAss.plot
    MAIN_DEPENDENCY plotMyAss.plot)
```

Cette commande crée une cible dont l'unique dépendance est le fichier plotMyAss.plot. Si ce dernier est modifié, le Makefile généré par CMake le detecte et re-copie le fichier dans le repertoire de build. Il faut également que plotMyAss.plot soit une dépendance de l'executable afin que la copie soit lancée à la compilation:

```
add_executable(MonExec ${SRC_FILES} plotMyAss.plot)
```

GESTION DE CONFIGURATIONS

Il est utile d'avoir plusieurs configurations de compilation pour gérer son projet, Release et Debug par exemple.
Une configuration définit les options à passer au compilateur.
Par exemple en Debug on aura l'option ”-g” qui permet d'ajouter les symboles de debug à l'executable généré.
En Release on aura l'option ”-O3” qui active les optimisations de niveau 3.

La bonne manière de gérer les configurations avec CMake est d'avoir un repertoire build par configuration:

- mon-projet-build-Release
- mon-projet-build-Debug
- mon-projet
  - CMakeLists.txt
  - ...

On utilise ensuite la variable CMAKE_BUILD_TYPE pour choisir sa configuration (CMake en propose plusieurs de base dont Debug et Release, mais il est possible d'en définir d'autres).
Dans CMakeLists.txt on ajoute (après project()):

```
if(NOT CMAKE_BUILD_TYPE)
  set(CMAKE_BUILD_TYPE "Release" CACHE STRING
      "Choose the type of build, options are: Debug Release."
endif(NOT CMAKE_BUILD_TYPE)
# Affiche la configuration activée:
message(${CMAKE_BUILD_TYPE})
```

Ainsi la configuration par défaut est Release.

GESTION DE CONFIGURATIONS

Ensuite on lance CMake dans chaque repertoire de build en activant la bonne configuration:

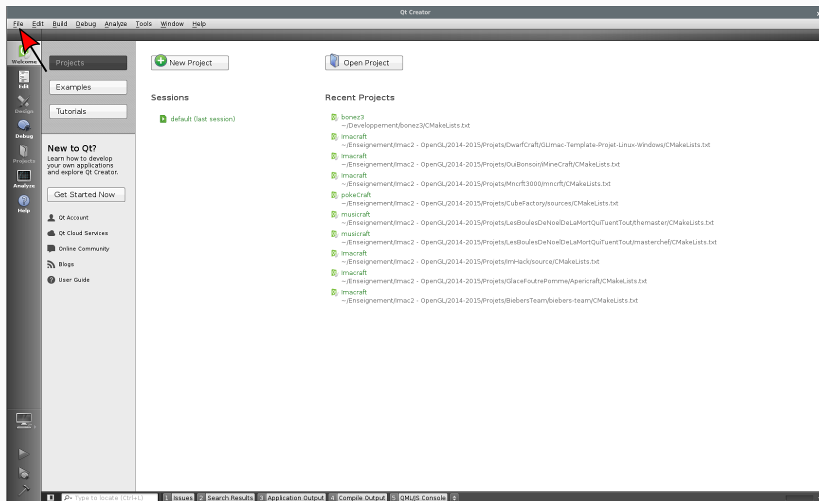
```
cd mon-projet-build-Release
cmake ../mon-projet -DCMAKE_BUILD_TYPE=Release
cd ../mon-projet-build-Debug
cmake ../mon-projet -DCMAKE_BUILD_TYPE=Debug
```

Inutile de respecifier la configuration lors du relancement de CMake dans chacun des repertoire: la variable est mise en cache dans CMakeCache.txt.

QTCREATOR

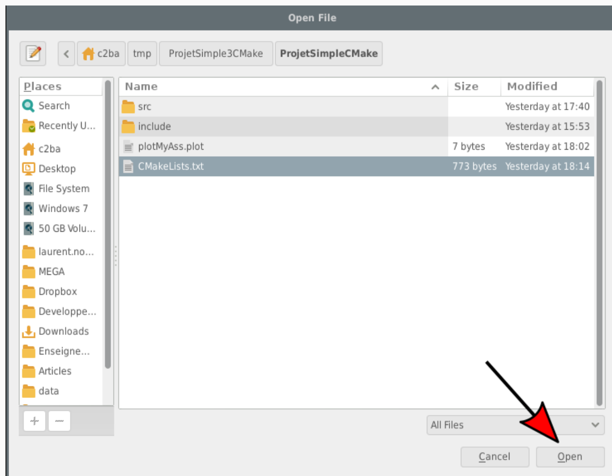
INTEGRATION AVEC QTCREATOR

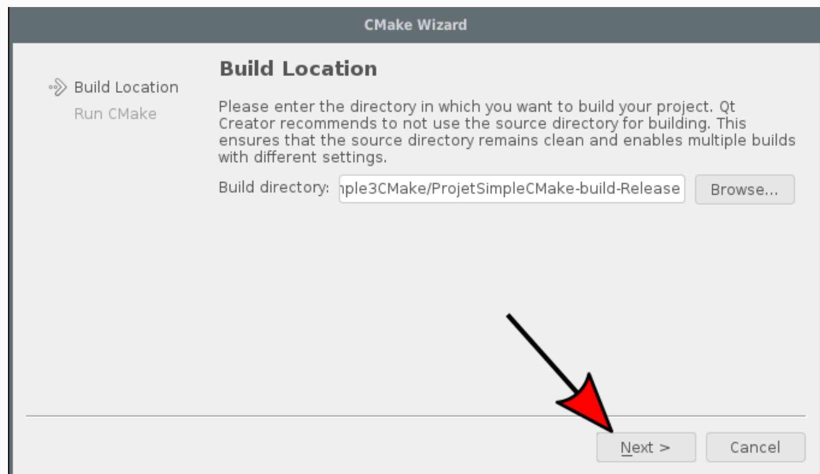
QtCreator est un IDE qui s'intègre très bien avec CMake: il interprete les fichiers CMakeLists.txt comme des projets.



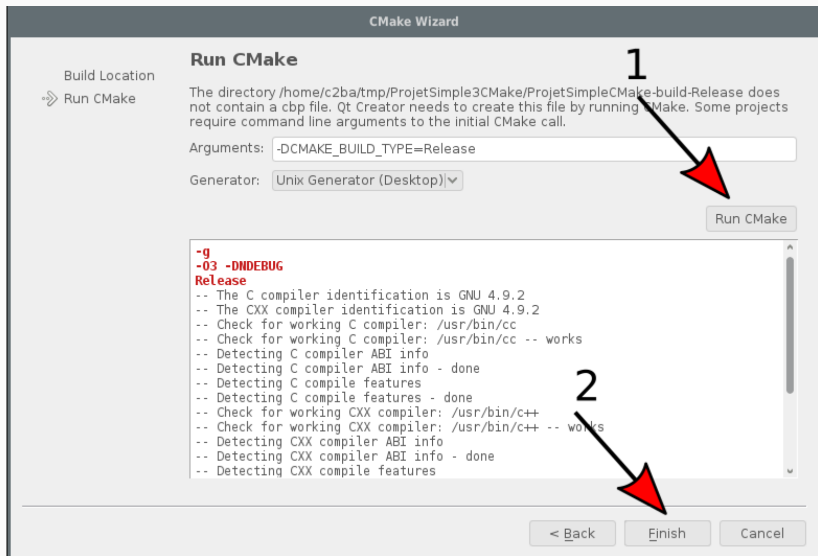
File → Open File or Project

SELECTIONNER LE FICHIER CMAKELISTS.TXT

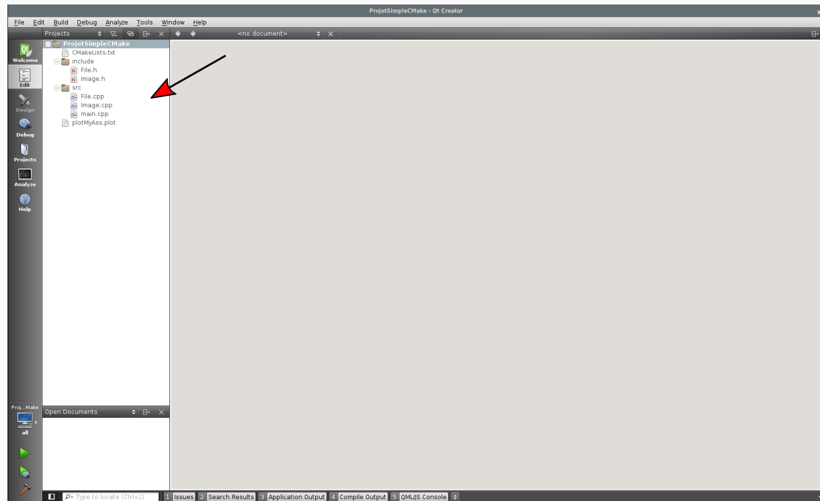



CMake Wizard
Build Location
Run CMake
Build Location
Please enter the directory in which you want to build your project. Qt Creator recommends to not use the source directory for building. This ensures that the source directory remains clean and enables multiple builds with different settings.
Build directory:
nple3CMake/ProjetSimpleCMake-build-Release
Browse...
Next >
Cancel

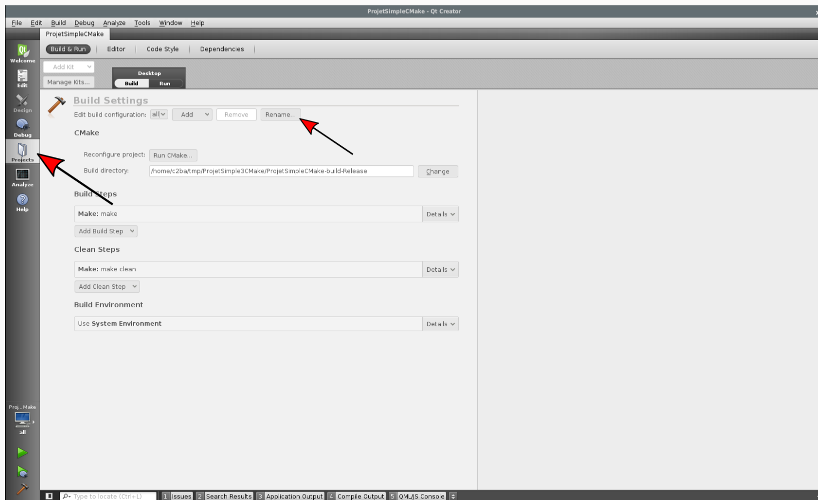
LANCER CMAKE AVEC SES ARGUMENT



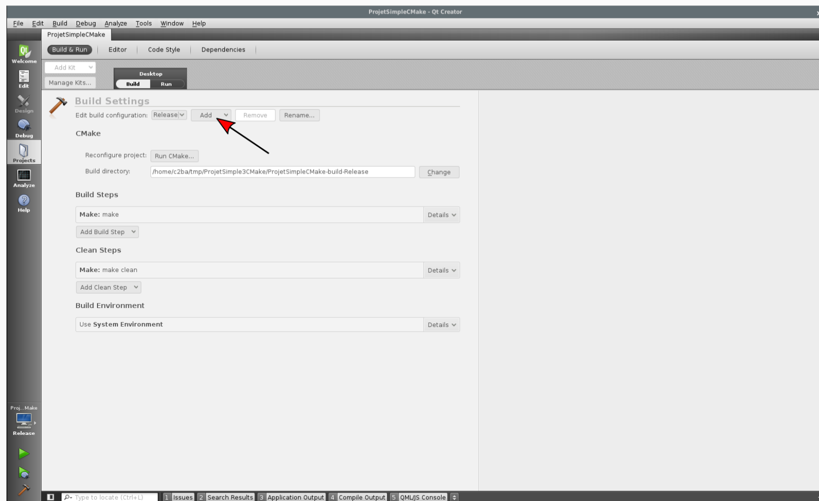
ARBORESCENCE DU PROJET



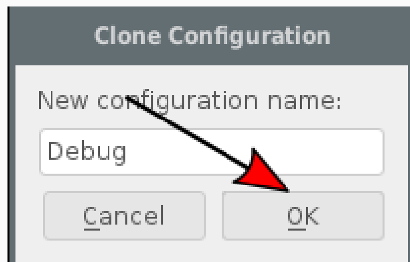
PANNEAU DE CONFIGURATION DU PROJET



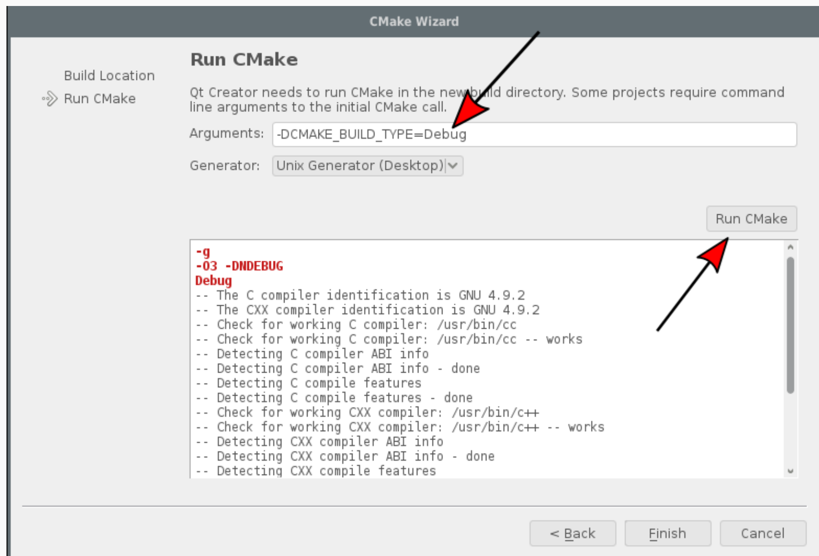
AJOUTER UNE CONFIGURATION



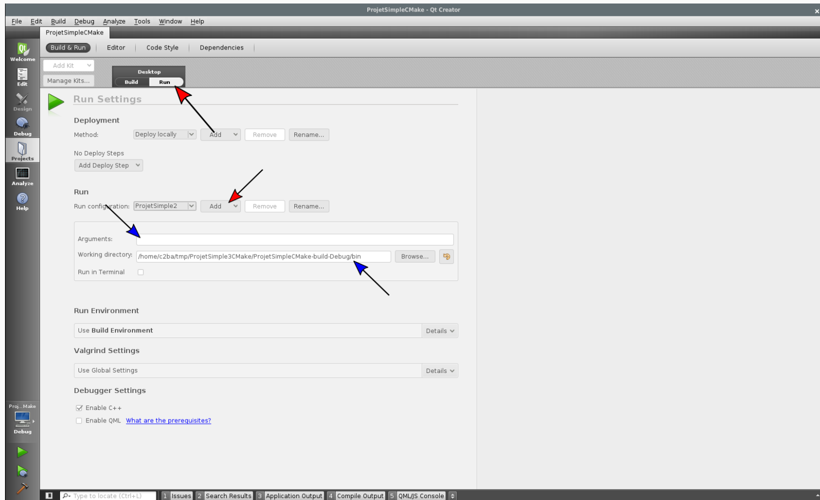
CHOISIR LE NOM DE LA CONFIGURATION



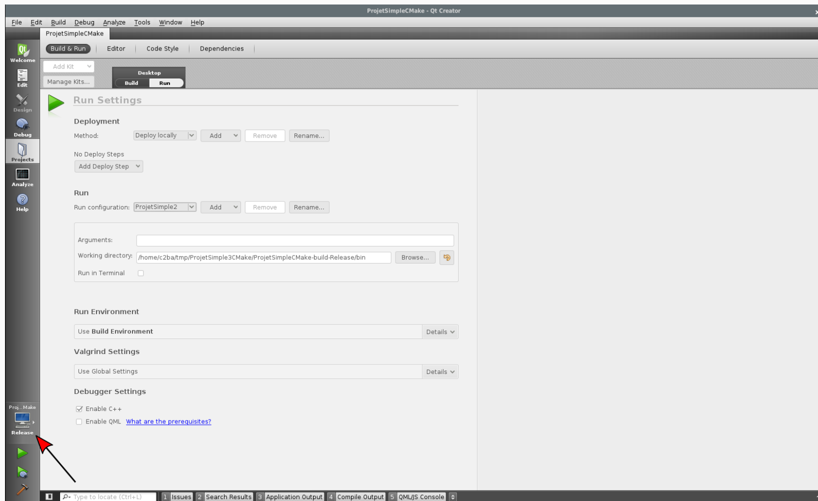
LANCER À NOUVEAU CMAKE



RUN SETTINGS



CHANGER RAPIDEMENT DE CONFIGURATION



Questions?