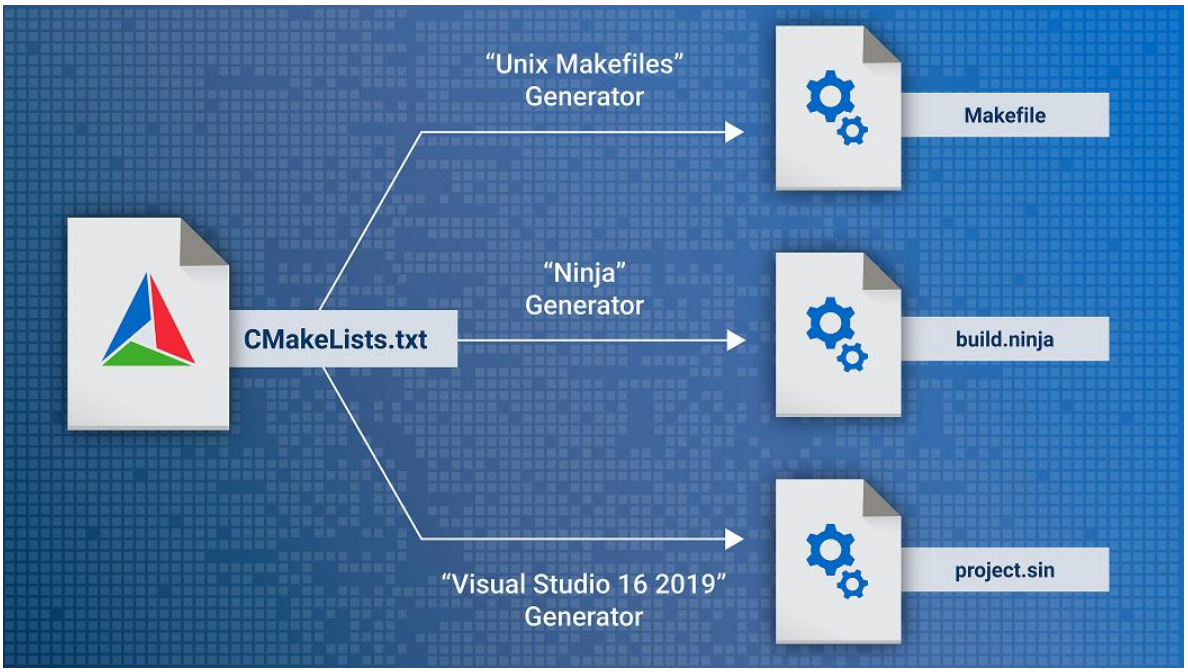
Structuration d'un projet et introduction à CMake

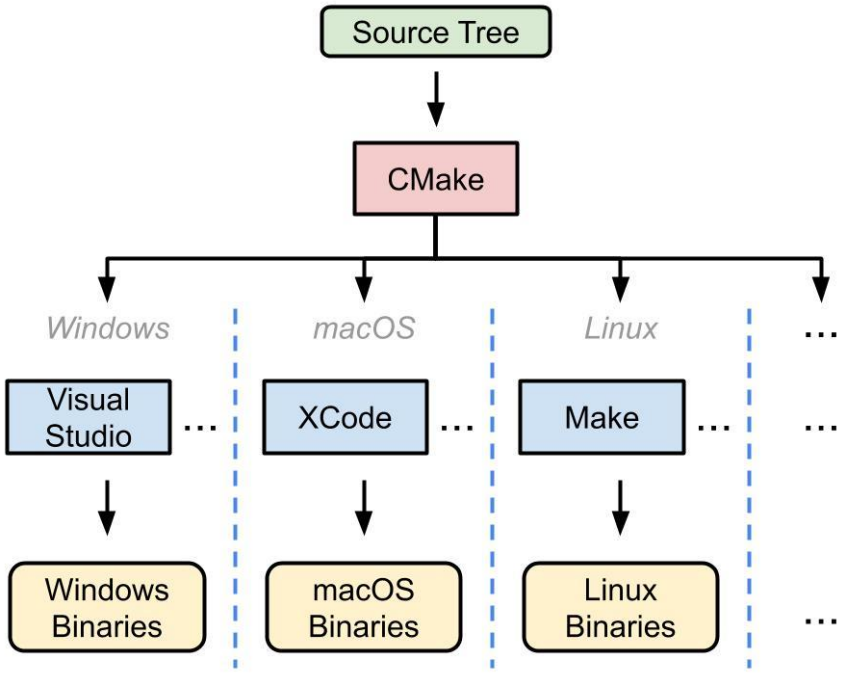
Introduction

CMake est un système de génération de makefiles et de projets destiné à simplifier la compilation de logiciels. Créé par Kitware, CMake permet de gérer des builds multi-plateformes de manière efficace et flexible. En utilisant des fichiers de configuration appelés **CMakeLists.txt**, CMake décrit la structure du projet, les dépendances, et les instructions de compilation, indépendamment de l'environnement ou du compilateur utilisé :



Caractéristiques

- **Multi-plateforme** : il supporte une grande variété de systèmes d'exploitation et de compilateur
- **Flexibilité** : il dispose de nombreuses options de configuration et peut être adapté à presque tous les types de projets
- **Gestion des dépendances** : il peut trouver et configurer automatiquement les bibliothèques externes nécessaires au projet
- **Intégration avec les IDE** : il peut générer des fichiers de projet pour différents IDE



A partir du code source (« Source Tree » càd fichiers .cpp, .c, .h, …), l'utilitaire **CMake** va, à partir du fichier de configuration **CMakeList.txt**, tout ce qui est nécessaire à la compilation du projet. Par exemple, sous Linux, il va générer le Makefile qui permettra de compiler et de générer les « binaries » (fichiers .o et exécutables).

Exemple introductif

Suppons nous trouver dans le répertoire **ProjetIntro** contenant le fichier **main.cpp** :

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Hello CMake !" << endl;
  return 0;
}
```

et le fichier **CMakeLists.txt** suivant :

```
cmake_minimum_required(VERSION 3.10)

# Nom du projet
project(HelloWorld)

# Ajouter un exécutable
```

```
add_executable(hello main.cpp)
```

Ce fichier CMake fait les choses suivantes :

1. Spécifie la version minimale de Cmake requise
2. Définit le nom du projet
3. Ajoute un exécutable nommé **hello** qui est construit à partir du fichier source **main.cpp**

L'exécution de **CMake** fournit :

```
[student@moon ProjetIntro]$ ls -l
total 8
-rw-rw-r--. 1 student student 131 11 jui 07:05 CMakeLists.txt
-rw-rw-r--. 1 student student 105 11 jui 07:03 main.cpp
[student@moon ProjetIntro]$ mkdir build
[student@moon ProjetIntro]$ cd build
[student@moon build]$ cmake ..
-- The C compiler identification is GNU 8.4.1
-- The CXX compiler identification is GNU 8.4.1
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to:
/home/student/CppTheorieCode/CMake/ProjetIntro/build
[student@moon build]$ ls -l
total 28
-rw-rw-r--. 1 student student 13888 11 jui 07:17 CMakeCache.txt
drwxrwxr-x. 5 student student   230 11 jui 07:17 CMakeFiles
-rw-rw-r--. 1 student student  1668 11 jui 07:17 cmake_install.cmake
-rw-rw-r--. 1 student student  5197 11 jui 07:17 Makefile
[student@moon build]$ make
Scanning dependencies of target hello
[ 50%] Building CXX object CMakeFiles/hello.dir/main.cpp.o
[100%] Linking CXX executable hello
[100%] Built target hello
[student@moon build]$ ls -l
total 48
-rw-rw-r--. 1 student student 13888 11 jui 07:17 CMakeCache.txt
drwxrwxr-x. 5 student student   230 11 jui 07:17 CMakeFiles
-rw-rw-r--. 1 student student  1668 11 jui 07:17 cmake_install.cmake
```

```
-rwxrwxr-x. 1 student student 18104 11 jui 07:17 hello
-rw-rw-r--. 1 student student  5197 11 jui 07:17 Makefile
[student@moon build]$ hello
Hello CMake !
[student@moon build]$
```

où on observe que

- Le fait de créer un répertoire **build** permet de séparer clairement le code source des fichiers résultant de la compilation
- La commande **cmake** sur le répertoire **..** lit le fichier **CMakeLists.txt** et crée un ensemble de fichiers et de répertoires nécessaires à la compilation du projet, dont notamment le fichier **Makefile**
- Il suffit alors de lancer la commande **make** qui va lire le fichier **Makefile** et réaliser la compilation

Structurer correctement son projet

Un projet multi-fichiers, et en particulier un projet CMake, est généralement structuré de manière à séparer les fichiers sources (.cpp), les entêtes (.h), et les fichiers de configuration.

Considérons l'exemple d'un projet contenant un programme **main.cpp** utilisant la classe **Person** constituée des deux fichiers **Person.h** et **Person.cpp**. Il est structuré selon

```
ProjetStructure/
├── CMakeLists.txt
├── include/
│   └── Person.h
├── src/
│   └── Person.cpp
└── main.cpp
```

où le fichier **Person.h** contient

```
#ifndef PERSON_H
#define PERSON_H

#include <iostream>
using namespace std;

class Person {
  private:
    string name;
```

```
    int age;

  public:
    Person(string name, int age);
    string toString() const;
};

#endif
```

et le fichier **Person.cpp** contient

```
#include "Person.h"
#include <sstream>
using namespace std;

Person::Person(string name, int age) {
  this->name = name;
  this->age = age;
}

string Person::toString() const {
  ostringstream ss;
  ss << name << " (" << age << ")";
  return ss.str();
}
```

Le fichier **CMakeLists.txt** contient quant à lui :

```
cmake_minimum_required(VERSION 3.10)

# Nom du projet
project(ProjetStructure)

# Spécifier le standard C++
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Inclure le répertoire include
include_directories(${PROJECT_SOURCE_DIR}/include)

# Ajouter les fichiers source
set(SOURCES
    src/Person.cpp
    main.cpp
)

# Créer l'exécutable
add_executable(${PROJECT_NAME} ${SOURCES})
```

où

- **set(CMAKE_CXX_STANDARD 11)** définit la version du C++ à utiliser

- **set(CMAKE_CXX_STANDARD_REQUIRED)** précise que la version du C++ spécifiée (ici la 11) est strictement requise. Si le compilateur ne supporte pas cette version, la configuration du projet échouera
- **project(ProjetStructure)** a défini le nom du projet mais également défini une variable **PROJECT_NAME** qui peut être utilisée autrepart dans le fichier CMakeLists.txt → dans cet exemple, cette variable va donner un nom à l'exécutable créé via la ligne « add_executable($(PROJECT_NAME) …)
- **include_directories(${PROJECT_SOURCE_DIR}/include)** ajoute le répertoire des entêtes au chemin d'inclusion. La variable **PROJECT_SOURCE_DIR** contient le chemin vers le répertoire du projet, celui-ci étant récupéré par la commande **cmake ..**
- **set(SOURCES …)** liste les fichiers sources du projet → en toute généralité la commande **set(…)** permet de définir des variables
- **add_executable(${PROJECT_NAME} ${SOURCES})** crée l'exécutable à partir des fichiers sources spécifiés

L'exécution de CMake fournit ici :

```
[student@moon ProjetStructure]$ ls -l
total 8
-rw-rw-r--. 1 student student 403 11 jui 08:46 CMakeLists.txt
drwxrwxr-x. 2 student student  22 11 jui 08:42 include
-rw-rw-r--. 1 student student 161 11 jui 08:02 main.cpp
drwxrwxr-x. 2 student student  24 11 jui 08:42 src
[student@moon ProjetStructure]$ mkdir build
[student@moon ProjetStructure]$ cd build
[student@moon build]$ cmake ..
-- The C compiler identification is GNU 8.4.1
-- The CXX compiler identification is GNU 8.4.1
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to:
/home/student/CppTheorieCode/CMake/ProjetStructure/build
[student@moon build]$ ls -l
```

```
total 28
-rw-rw-r--. 1 student student 13923 11 jui 09:05 CMakeCache.txt
drwxrwxr-x. 5 student student   240 11 jui 09:05 CMakeFiles
-rw-rw-r--. 1 student student  1676 11 jui 09:05 cmake_install.cmake
-rw-rw-r--. 1 student student  6204 11 jui 09:05 Makefile
[student@moon build]$ make
Scanning dependencies of target ProjetStructure
[ 33%] Building CXX object
CMakeFiles/ProjetStructure.dir/src/Person.cpp.o
[ 66%] Building CXX object CMakeFiles/ProjetStructure.dir/main.cpp.o
[100%] Linking CXX executable ProjetStructure
[100%] Built target ProjetStructure
[student@moon build]$ ls -l
total 52
-rw-rw-r--. 1 student student 13923 11 jui 09:05 CMakeCache.txt
drwxrwxr-x. 5 student student   240 11 jui 09:05 CMakeFiles
-rw-rw-r--. 1 student student  1676 11 jui 09:05 cmake_install.cmake
-rw-rw-r--. 1 student student  6204 11 jui 09:05 Makefile
-rwxrwxr-x. 1 student student 24000 11 jui 09:05 ProjetStructure
[student@moon build]$ ProjetStructure
Person = wagner (50)
[student@moon build]$
```

Créer sa propre librairie

Les commandes de base de **CMake** que nous connaissons déjà sont

- **cmake_minimum_required**
- **project**
- **add_executable**

L'idée à présent est de créer une librairie rassemblant des classes ayant un thème commun. Cette librairie sera traitée et compilée séparément, puis ensuite liée à l'exécutable. Pour cela, nous aurons besoin des 2 nouvelles commandes suivantes :

- **add_library** → créé une bibliothèque à partir d'une liste de fichiers source. Les librairies peuvent être statiques (STATIC) ou partagées (SHARED). Une bibliothèque statique est embarquée à la compilation dans le fichier exécutable pour ne former plus qu'un seul fichier contenant le programme et sa librairie. Dans le cas d'une librairie partagée, celle-ci n'est pas embarquée dans le même fichier que l'exécutable. Elle est chargée en mémoire au moment de l'exécution.

- **target_link_libraries** → lie une cible (l'exécutable en général) à une ou plusieurs librairies. Elle est essentielle pour résoudre les dépendances entre les différents composants du projet

Reprenons le même exemple que précédemment, avec les mêmes fichiers sources et la même structure de répertoires. La seule différence est le contenu du fichier **CMakeLists.txt** :

```
cmake_minimum_required(VERSION 3.10)

# Nom du projet
project(ProjetStructure)

# Spécifier le standard C++
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Inclure le répertoire include
include_directories(${PROJECT_SOURCE_DIR}/include)

# Créer une bibliothèque statique
add_library(MaLibrairie STATIC src/Person.cpp)

# Créer l'exécutable
add_executable(${PROJECT_NAME} main.cpp)

# Lier l'exécutable à la bibliothèque
target_link_libraries(${PROJECT_NAME} MaLibrairie)
```

L'exécution de cmake fournit :

```
[student@moon ProjetLibrairie]$ ls -l
total 8
-rw-rw-r--. 1 student student 501 11 jui 09:44 CMakeLists.txt
drwxrwxr-x. 2 student student  22 11 jui 09:25 include
-rw-rw-r--. 1 student student 161 11 jui 09:25 main.cpp
drwxrwxr-x. 2 student student  24 11 jui 09:25 src
[student@moon ProjetLibrairie]$ mkdir build
[student@moon ProjetLibrairie]$ cd build
[student@moon build]$ cmake ..
-- The C compiler identification is GNU 8.4.1
-- The CXX compiler identification is GNU 8.4.1
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
```

```
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to:
/home/student/CppTheorieCode/CMake/ProjetLibrairie/build
[student@moon build]$ ls -l
total 28
-rw-rw-r--. 1 student student 13923 11 jui 09:46 CMakeCache.txt
drwxrwxr-x. 6 student student   263 11 jui 09:46 CMakeFiles
-rw-rw-r--. 1 student student  1676 11 jui 09:46 cmake_install.cmake
-rw-rw-r--. 1 student student  6646 11 jui 09:46 Makefile
[student@moon build]$ make
Scanning dependencies of target MaLibrairie
[ 25%] Building CXX object CMakeFiles/MaLibrairie.dir/src/Person.cpp.o
[ 50%] Linking CXX static library libMaLibrairie.a
[ 50%] Built target MaLibrairie
Scanning dependencies of target ProjetStructure
[ 75%] Building CXX object CMakeFiles/ProjetStructure.dir/main.cpp.o
[100%] Linking CXX executable ProjetStructure
[100%] Built target ProjetStructure
[student@moon build]$ ls -l
total 60
-rw-rw-r--. 1 student student 13923 11 jui 09:46 CMakeCache.txt
drwxrwxr-x. 6 student student   263 11 jui 09:46 CMakeFiles
-rw-rw-r--. 1 student student  1676 11 jui 09:46 cmake_install.cmake
-rw-rw-r--. 1 student student  4660 11 jui 09:46 libMaLibrairie.a
-rw-rw-r--. 1 student student  6646 11 jui 09:46 Makefile
-rwxrwxr-x. 1 student student 24000 11 jui 09:46 ProjetStructure
[student@moon build]$ ProjetStructure
Person = wagner (50)
[student@moon build]$
```

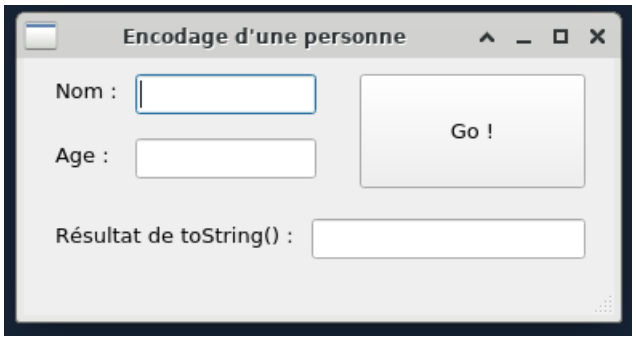
Gérer les dépendances et utiliser des librairies externes

Le plus intéressant est de pouvoir inclure des librairies externes facilement sans devoir se soucier des options de compilation, de l'emplacement des fichiers de ces librairies.

Nous reprenons à nouveau l'exemple précédent mais cette fois-ci, nous introduisons une fenêtre graphique Qt représentée par la classe **MainWindowsPerson**. La structure du projet est alors la suivante :

```
ProjetQt/
├── CMakeLists.txt
├── include/
│   └── mainwindowperson.h
│   └── Person.h
│   └── ui_mainwindowperson.h
└── src/
    └── main.cpp
    └── mainwindowperson.cpp
    └── moc_mainwindowperson.cpp
    └── Person.cpp
```

où les fichiers **ui_mainwindowperson.h** et **moc_mainwindowperson.cpp** n'ont pas grand intérêt car ils ont été générés automatiquement par QtCreator lors de la conception « drag and grop » de la fenêtre graphique :



Par contre, voici le contenu du fichier **mainwindowperson.h** :

```
#ifndef MAINWINDOWPERSON_H
#define MAINWINDOWPERSON_H

#include <QMainWindow>

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindowPerson; }
QT_END_NAMESPACE

class MainWindowPerson : public QMainWindow
{
    Q_OBJECT

public:
    MainWindowPerson(QWidget *parent = nullptr);
    ~MainWindowPerson();

private slots:
    void on_pushButtonGo_clicked();
```

```
private:
    Ui::MainWindowPerson *ui;
};
#endif // MAINWINDOWPERSON_H
```

Et celui du fichier **mainwindowperson.cpp** :

```
#include "mainwindowperson.h"
#include "ui_mainwindowperson.h"
#include "Person.h"

MainWindowPerson::MainWindowPerson(QWidget *parent) :
QMainWindow(parent), ui(new Ui::MainWindowPerson) {
    ui->setupUi(this);
}

MainWindowPerson::~MainWindowPerson() {
    delete ui;
}

void MainWindowPerson::on_pushButtonGo_clicked() {
    string name = ui->lineEditName->text().toStdString();
    int age = stoi(ui->lineEditAge->text().toStdString());

    Person p(name,age);
    ui->lineEdit->setText(QString::fromStdString(p.toString()));
}
```

Le fichier **CMakeLists.txt** quant à lui est

```
cmake_minimum_required(VERSION 3.10)

# Nom du projet
project(ApplicPerson)

# Spécifier le standard C++
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Trouver les modules Qt nécessaires
find_package(Qt5 COMPONENTS Core Gui Widgets REQUIRED)

# Inclure le répertoire include
include_directories(include)

# Ajouter les fichiers sources automatiquement
file(GLOB SOURCES "src/*.cpp")
```

```
# Ajouter l'exécutable
add_executable(${PROJECT_NAME} ${SOURCES})

# Lier les bibliothèques Qt nécessaires
target_link_libraries(${PROJECT_NAME} Qt5::Core Qt5::Gui Qt5::Widgets)
```

où

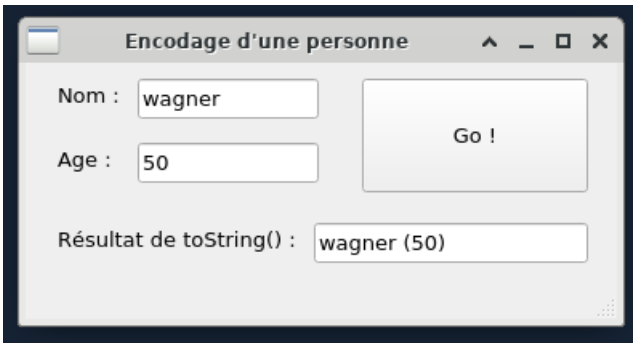
- **find_package(Qt5 COMPONENTS Core Gui Widgets REQUIRED)** permet de trouver les modules de la librairie Qt nécessaires au projet
- **file(GLOB SOURCES "src/*.cpp")** permet d'incorporer dans la variable SOUCES tous les fichiers source .cpp situés dans le répertoire src, sans devoir les incorporer un à un
- **target_link_libraries(${PROJECT_NAME} Qt5::Core Qt5::Gui Qt5::Widgets)** lie les librairies Qt à l'exécutable

L'exécution de cmake fournit :

```
[student@moon ProjetQt]$ ls -l
total 4
-rw-rw-r--. 1 student student 589 11 jui 10:29 CMakeLists.txt
drwxrwxr-x. 2 student student  77 11 jui 10:23 include
drwxrwxr-x. 2 student student 100 11 jui 10:23 src
[student@moon ProjetQt]$ mkdir build
[student@moon ProjetQt]$ cd build
[student@moon build]$ cmake ..
-- The C compiler identification is GNU 8.4.1
-- The CXX compiler identification is GNU 8.4.1
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to:
/home/student/CppTheorieCode/CMake/ProjetQt/build
[student@moon build]$ ls -l
total 32
-rw-rw-r--. 1 student student 14313 11 jui 11:02 CMakeCache.txt
drwxrwxr-x. 5 student student   237 11 jui 11:02 CMakeFiles
-rw-rw-r--. 1 student student  1662 11 jui 11:02 cmake_install.cmake
```

```
-rw-rw-r--. 1 student student  8332 11 jui 11:02 Makefile
[student@moon build]$ make
Scanning dependencies of target ApplicPerson
[ 20%] Building CXX object CMakeFiles/ApplicPerson.dir/src/Person.cpp.o
[ 40%] Building CXX object CMakeFiles/ApplicPerson.dir/src/main.cpp.o
[ 60%] Building CXX object
CMakeFiles/ApplicPerson.dir/src/mainwindowperson.cpp.o
[ 80%] Building CXX object
CMakeFiles/ApplicPerson.dir/src/moc_mainwindowperson.cpp.o
[100%] Linking CXX executable ApplicPerson
[100%] Built target ApplicPerson
[student@moon build]$ ls -l
total 88
-rwxrwxr-x. 1 student student 56048 11 jui 11:02 ApplicPerson
-rw-rw-r--. 1 student student 14313 11 jui 11:02 CMakeCache.txt
drwxrwxr-x. 5 student student   237 11 jui 11:02 CMakeFiles
-rw-rw-r--. 1 student student  1662 11 jui 11:02 cmake_install.cmake
-rw-rw-r--. 1 student student  8332 11 jui 11:02 Makefile
[student@moon build]$ ApplicPerson
[student@moon build]$
```

tandis qu'un clic sur le bouton « Go ! » de la fenêtre à fourni



Le clic sur le bouton provoque l'exécution de la méthode **on_pushButtonGo_clicked()** qui

- récupère le nom et l'age d'une personne encodée par l'utilisateur
- instancie un objet de la classe **Person**
- affiche dans l'interface graphique le résultat de la méthode **toString()** de l'objet créé

Il faut noter que la librairie Qt doit être installée sur le système pour que **CMake** puisse correctement générer les fichiers de configuration du projet.

Remarque

Il est encore possible de créer plusieurs exécutables différents, utilisant des modules communs. Mais cela sort du cadre de cette introduction à CMake.