Introduction à *CMake*

Michel Simatic

Télécom SudParis

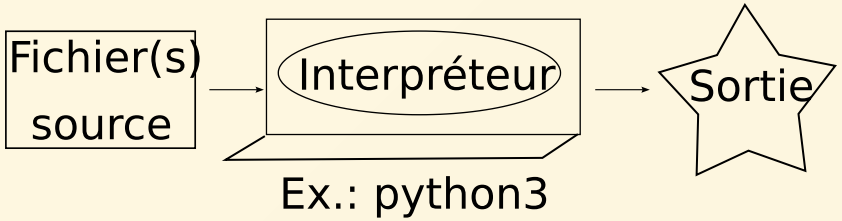
14 mai 2025

Plan

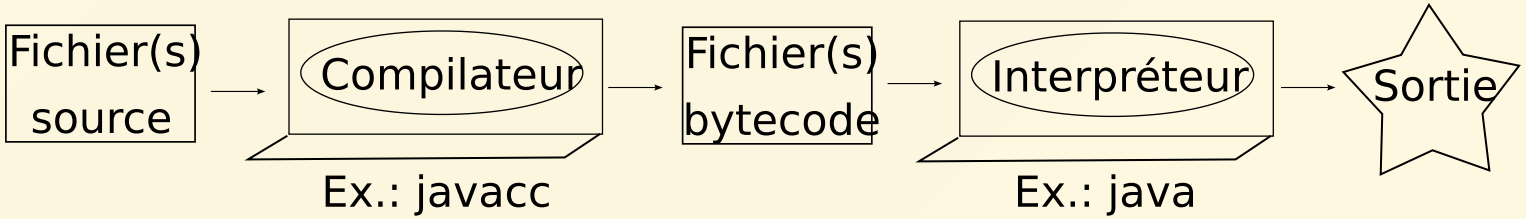
Objectif de *CMake*

Rappels ? Types(/familles) de langages informatiques

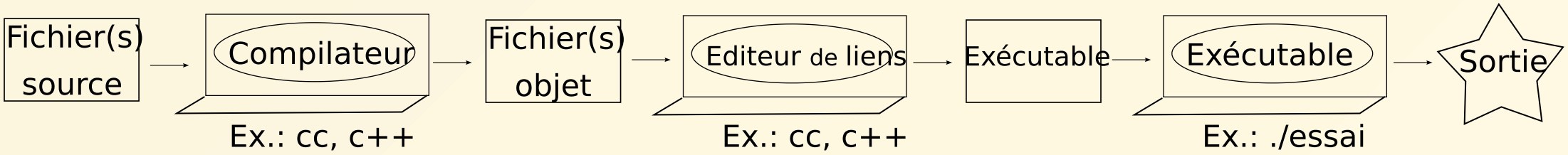
- Langages interprétés (Python, PHP, LUA...) : + Vitesse de dév. / - Perf.



- Langages semi-compilés (Java, C#...) : Le meilleur des mondes ?



- Langages compilés (C, C++, RUST...) : + Perf. / - Vitesse de dév.



Zoom sur compromis Vitesse de dév. / Perf (1/2)

Performances

Considérons le programme Python suivant (tiré de ce TP JIN 3A)

```
nb_count = 100000000
counter = nb_count
i = nb_count
while i > 0 :
    counter -= 1
    i -= 1
print('counter = ', counter)
```

Sur une machine avec processeur i7 à 2,80 Ghz et 32 Go de RAM :

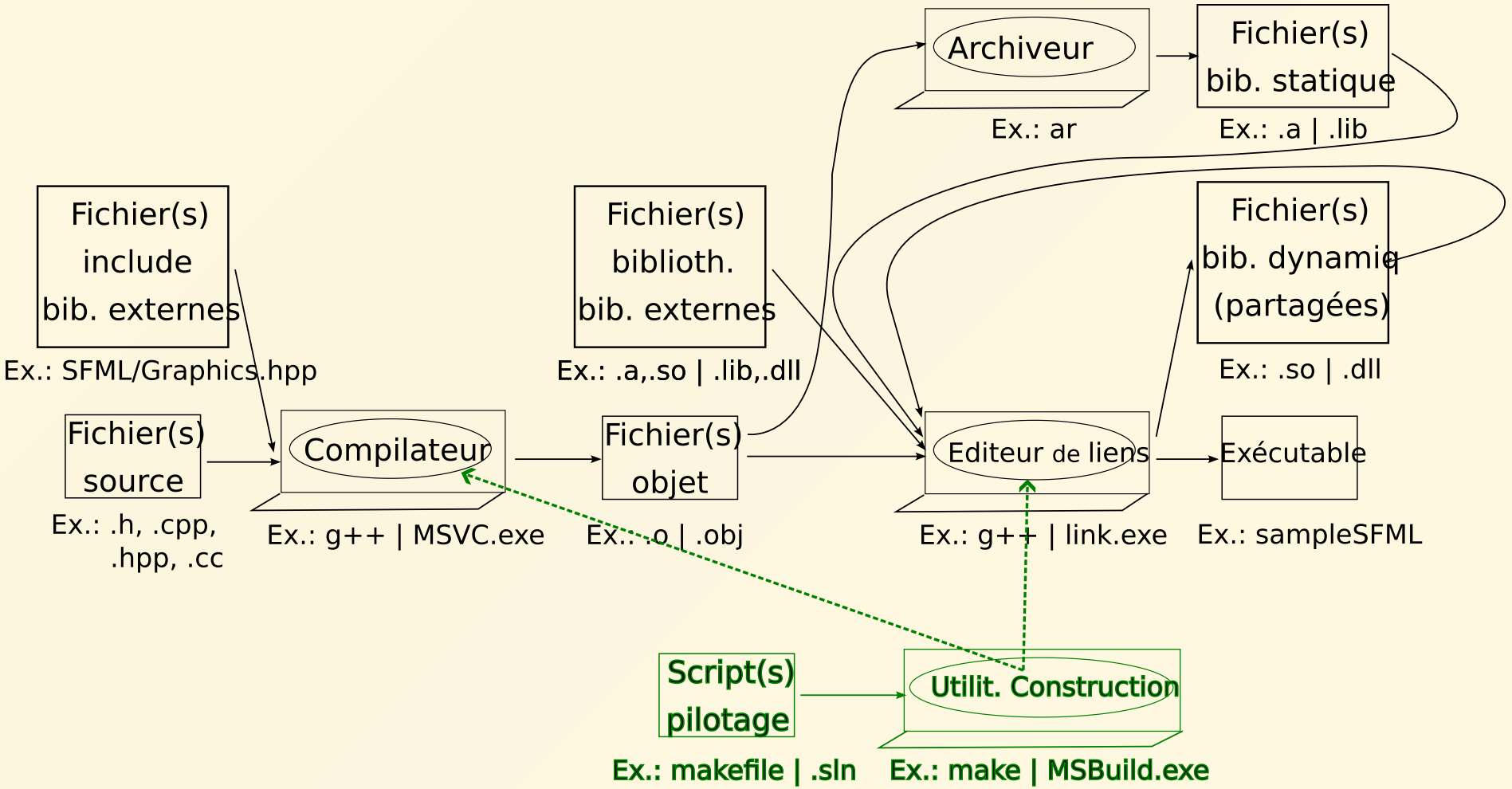
- Ce programme s'exécute en 21,9 secondes et consomme 21,8 secondes de CPU.
- La version C++ de ce programme s'exécute en 73 millisecondes avec 78 millisecondes de CPU consommée.
- $\Rightarrow$ Ce programme Python est 300 fois plus lent que son équivalent C++.

Zoom sur compromis Vitesse de dév. / Perf (2/2)

Vitesse de développement

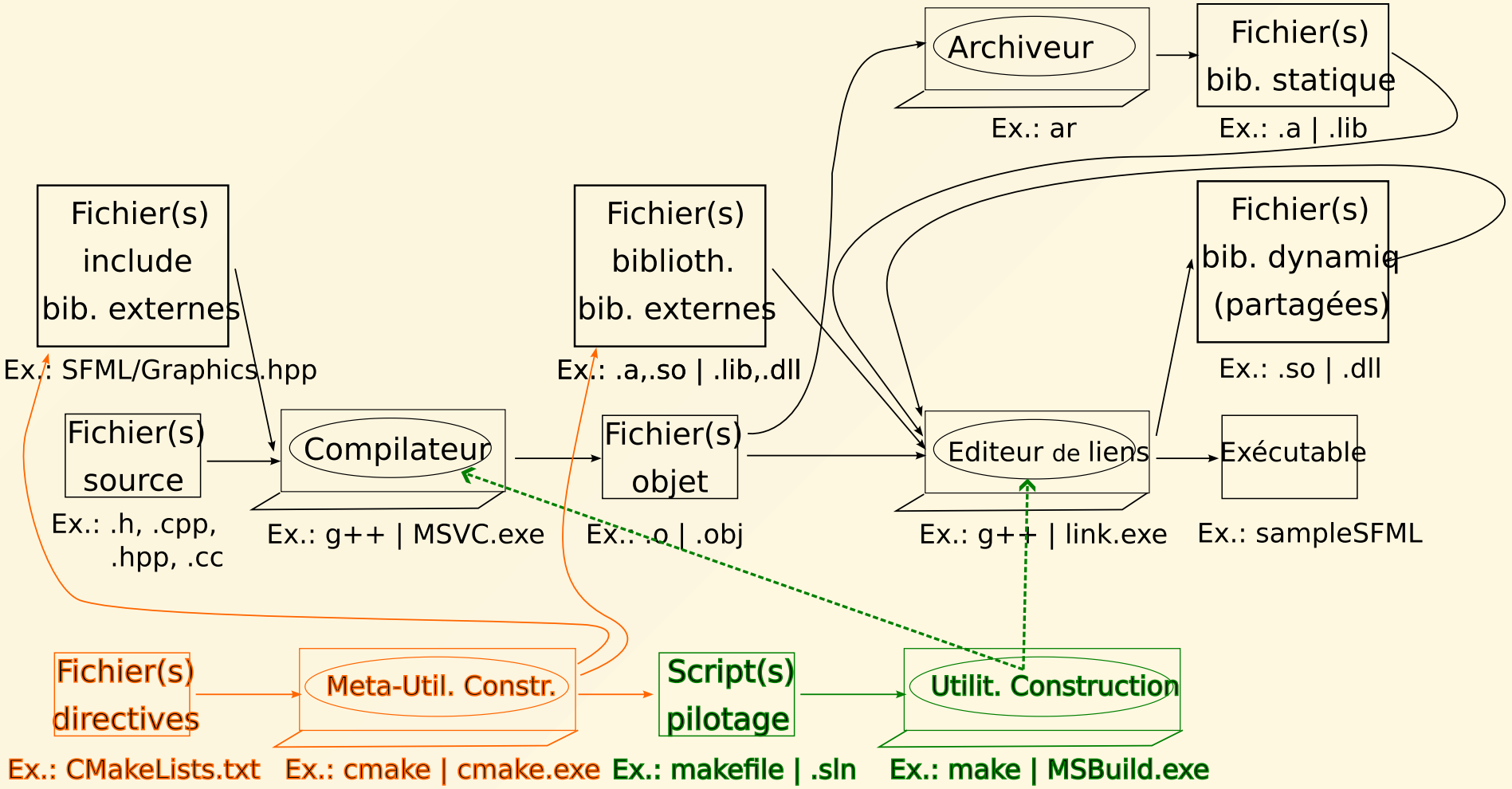
- YouTube a initialement été développé en Python. Pour améliorer ses performances, il a ensuite été porté en C++.
- Facebook est développé en PHP. Pour améliorer ses performances, *Meta* a développé un compilateur PHP $\rightarrow$ C++.
- Dans le monde du jeu vidéo :
  - En 2023, objectif stage de fin d'étude d'une étudiante JIN = Améliorer la parallélisation, sur les différentes machines des développeurs·ses des modules C++ d'un jeu.
  - Un grand classique :
    - Le cœur du jeu est écrit en C++ (langage compilé).
    - Les règles du jeu son implantées en LUA (langage interprété).

Rappels ? Chaîne de compilation



NB : Le compilateur fait le prétraitement (*preprocessing*), la compilation et l'optimisation.

Objectif de *CMake* (sous forme de schéma)



Objectif de *CMake* (sous forme de discours)

- *CMake* est un méta-« utilitaire de construction » destiné à générer, pour la plupart des systèmes d'exploitation existants (Windows Linux, macOS, Windows…), des scripts de pilotage pour des « utilitaires de construction » de logiciel: *make*, *MSBuild*, *Ninja*
- Il facilite la vie des développeurs·ses :
  - Il limite les besoins d'installation de bibliothèques avant de pouvoir compiler le logiciel
  - Il évite de se préoccuper des soucis de configuration d'outil de construction de logiciel, compilateur et éditeur de liens.
    - En particulier, il facilite le développement d'une version *debug* ou *production*.
  - Il permet de déclencher, sur un simple *commit* Git, la compilation du logiciel avec différents compilateurs (MSVC, gcc, clang) avec, par exemple, Microsoft Azure Pipelines (notion d'intégration continue).
  - Il garantit que les différentes bibliothèques requises pour la construction d'un logiciel sont aux bonnes versions.

Mini-TP

1. Objectif de *CMake*
2. → Mini-TP
3. Principales notions
4. Trucs et astuces

Mini-TP (1/3) : Déroulez les instructions suivantes

- Décompressez l'archive SampleSFML.zip dans le répertoire de votre choix (par exemple, dans `C:\users\votre_login\CSC4526`).
- Puis, dans un terminal (NB : Sous *Windows*, si les instructions ci-dessous ne fonctionnent pas avec un *Powershell* simple, exécutez la commande `& 'C:\Program Files\Microsoft Visual Studio\2022\Community\Common7\Tools\Launch-VsDevShell.ps1'`, cf. cette doc) :

```
cd cheminVersLeRepertoireSampleSFML
cmake --help # Pour voir tous les utilitaires de génération (générateurs) disponibles sur votre système
cmake -B buildM # (ou `cmake -BbuildM`) Pour que, dans buildM (M comme Manual), les scripts de pilotage
                # soient générés et les bibliothèques téléchargées.
cmake --build buildM # Pour déclencher toutes les générations avec l'utilitaire de génération
                     # (`cmake --build buildM --verbose` pour voir les commandes de compilation)
#
# La suite dépend de votre OS
#
# Sous Linux
./buildM/src/main/sampleSFML
# Sous macOS
./buildM/bin/src/main/sampleSFML
# Sous Windows
.\buildM\src\main\Debug\sampleSFML.exe
#
# Si votre projet contient des tests unitaires
#
cd buildM
ctest # Pour exécuter tous vos tests unitaires.
ctest --rerun-failed --output-on-failure # Pour re-exécuter de manière verbeuse les tests échoués.
```

Mini-TP (2/3) : Remarques sur les instructions déroulées précédemment

- Si vous modifiez un fichier dans `src`, `cmake --build buildM` ne régénère que les fichiers concernés par les modifications de ce fichier (essayez de faire un changement dans `src/main/main.cpp` pour le vérifier).
- Le répertoire `buildM/_deps` contient le résultat de la récupération des sources *SFML* à partir du dépôt https://github.com/SFML/SFML.git (cf. fichier `CMakeLists.txt` racine de SampleSFML.zip).
- Tout le travail de construction du logiciel a été fait dans ce répertoire `buildM`. Il suffit d'effacer ce répertoire pour s'en débarrasser.

Mini-TP (3/3) : Le *Build* réalisé peut coexister avec un autre *build*

- Avec votre *IDE*, appliquez la procédure [Construire un projet C++ avec *cmake*](../OutilsCSC4526/outilsCSC4526.html#construire-un-projet-c-avec-cmake).
  - *CLion* : Vous vous retrouvez avec des répertoires `cmake-build-debug` (contenant la génération par *CMake*) et `.idea` (contenant des information internes à *CLion*).
  - *Visual Studio* : Vous vous retrouvez avec des répertoires `out` (contenant la génération par *CMake*) et `.vs` (contenant des information internes à *Visual Studio*).
  - *Visual Studio Code* : Vous vous retrouvez avec des répertoires `build` (contenant la génération par *CMake*) et `.vscode` (contenant des information internes à *Visual Studio*).
- L'exécutable que vous venez de générer et celui généré au transparent précédent peuvent cohabiter sans problème.

Principales notions

1. Objectif de *CMake*
2. Mini-TP
3. → Principales notions
4. Trucs et astuces

Structure des projets dans l'UV *CSC4526*

Dans l'UV *CSC4526*, nous avons choisi d'appliquer systématiquement l'arborescence suivante :

- `doc` contient l'éventuelle documentation de votre projet.
- `resources` contient tous les fichiers ressource de votre projet (data, images, sons, fontes...).
- `src` contient tous les sources.
  - `core` contient l'âme, la moelle de votre projet. Ses fichiers servent à construire une bibliothèque que l'éditeur de liens liera avec le code de votre `main()` (respectivement vos tests unitaires) pour créer votre exécutable (respectivement l'exécutable de vos tests unitaires).
  - `main` contient un fichier définissant la fonction `main()` de votre projet.
  - `test` contient tous les tests unitaires de votre projet.

Seule exception = TP Fil rouge (SimpleGame) où l'arborescence respecte grosso modo celle des sources du livre *“SFML game development : learn how to use SFML 2.0 to develop your own feature-packed game”* de Jan Haller, Henrik Vogelius Hansson et Artur Moreira dont sont inspirés les sources de ce TP.

`CMakeLists.txt` global (1/2)

- Ce fichier (présent dans le répertoire racine du projet) contient les informations communes à toutes les entités logicielles (bibliothèques ou exécutables) de votre projet (cf. `SampleSFML/CMakeLists.txt` de SampleSFML.zip)
- `cmake_minimum_required(VERSION 3.26)` spécifie la version minimale de *CMake* qui doit lire votre `CMakeLists.txt`.
- `include(FetchContent)` spécifie qu'on utilise le module *CMake* `FetchContent`.
- `project(JIN4_SampleSFML VERSION 1.0.0 LANGUAGES CXX)` spécifie le nom de votre projet dans l'outil de construction (cf. `JIN4_SampleSFML.sln`), sa version et les langages informatiques utilisés pour votre projet.
- `IF(WIN32)`, `else()` et `endif()` permettent une génération conditionnelle (en fonction, par exemple, de l'OS utilisé).
- `set (CMAKE_EXPORT_COMPILE_COMMANDS ON)` permet de positionner un flag de génération *CMake* dont *SonarQube* a besoin, sous Windows, pour travailler correctement après.

`CMakeLists.txt` global (2/2)

- `set (BUILD_SHARED_LIBS FALSE)` indique à *CMake* de générer des bibliothèques statiques (et non dynamiques). Ainsi, même si les exécutables résultant sont plus gros, il n'y a rien à configurer au niveau du système d'exploitation des développeurs·ses pour que l'exécution puisse se faire correctement.
- `FetchContent_Declare` spécifie comment récupérer une bibliothèque externe (son repository GIT et la branche/tag de version à utiliser ou bien l'URL où récupérer une archive).
- `FetchContent_MakeAvailable` récupère effectivement une bibliothèque externe.
- `set(CMAKE_CXX_STANDARD 23)` spécifie que *CMake* doit dire au compilateur d'autoriser les spécificités du C++20 (ce qui inclut C++11, C++14, C++17, C++20, C++23, mais pas C++26).
- `add_subdirectory(src/main)` ajoute le sous-répertoire `src/main` à la construction.

Digression sur `FetchContent_Declare` (1/2)

Cet article explique qu'il est beaucoup plus performant de récupérer un projet via l'URL d'une archive que via un dépôt GIT (qui fait un `git clone`). Dit autrement, préférez:

```
FetchContent_Declare(
  sfml
  URL https://github.com/SFML/SFML/archive/refs/tags/3.0.0.tar.gz
)
```

à

```
FetchContent_Declare(
        sfml
        GIT_REPOSITORY https://github.com/SFML/SFML.git
        GIT_TAG 3.0.0
)
```

Digression sur `FetchContent_Declare` (2/2)

Les tests que nous avons faits sur la bibliothèque cereal confirment cette opinion :

- `FetchContent_Declare` avec paramètre `GIT_REPOSITORY` ==> elapsed = 2.02 secondes ; CPU = 3.02 secondes
- `FetchContent_Declare` avec paramètre `URL` ==> elapsed = 0.65 seconde ; CPU ~ 0 secondes

Seuls bémols sur l'utilisation d'une URL par rapport à l'utilisation d'un dépôt GIT :

1. Le `FetchContent_Declare` doit être précédé par
   ```
   cmake_policy(SET CMP0135 NEW) # This cmake_policy avoids warning by cmake when we fetch contents based on URL
   ```
2. Sous Linux, faire le fetch de *SFML* avec une `URL` entraîne des temps de génération 50-100% plus longs qu'avec le `GIT_REPOSITORY`.

Digression sur `enable_testing()` (quand votre projet contient des tests unitaires)

Comme dans `SixQuiPrend/CMakeList.txt` de SixQuiPrend.zip, `enable_testing()` spécifie qu'il y aura des fichiers pour les tests dans le répertoire courant et ses sous-répertoires.

NB : Pour que l'explorateur de tests de *Visual Studio* travaille correctement, cette instruction doit absolument :

1. apparaître dans le `CMakeLists.txt` de plus haut niveau (cf. documentation cmake),
2. être précédée par l'instruction `include(GoogleTest)` dans le `CMakeLists.txt` de plus haut niveau (cf. ce ticket).

`CMakeLists.txt` par entité logicielle (1/2)

- Le fichier `CMakeLists.txt` présent dans le répertoire contenant les sources de votre entité logicielle (exécutable ou bibliothèque) contient les informations permettant de spécifier comment construire votre entité logicielle (cf. `SampleSFML/src/main/CMakeLists.txt` de SampleSFML.zip).
- `add_executable(sampleSFML main.cpp)` spécifie que l'entité logicielle à générer est un exécutable du nom de `sampleSFML` construit avec tous les fichiers listés (seulement `main.cpp` dans le cas présent).
- `target_link_libraries(sampleSFML PUBLIC sfml-graphics)` spécifie que, pour construire votre entité logicielle `sampleSFML`, vous aurez besoin de faire l'édition de liens avec la bibliothèque `sfml-graphics` (et donc, implicitement, que vous aurez besoin des include *SFML* pour la compilation de vos `.cpp`).
- FYI Il est possible de construire la liste des sources de votre exécutable avec `file(GLOB SOURCES CONFIGURE_DEPENDS #.h #.cpp)` Mais certain·es "pro" de *cmake* déconseillent d'utiliser cette instruction. Iels disent qu'il vaut mieux lister explicitement les fichiers.

`CMakeLists.txt` par entité logicielle (2/2)

- Construction d'une bibliothèque (cf. `SixQuiPrend/src/core/CMakeList.txt` de SixQuiPrend.zip) :
  - `add_library(lib_core simulateur.cpp simulateur.h)` spécifie que l'entité logicielle à générer est une bibliothèque de nom `lib_core`, constituée des fichiers `simulateur.cpp` et `simulateur.h`.
  - `target_include_directories(lib_core PUBLIC ./)` spécifie qu'un source dans un autre répertoire pourra faire `#include` de fichiers situés dans le présent répertoire.

Trucs et astuces

1. Objectif de *CMake*
2. Mini-TP
3. Principales notions
4. → Trucs et astuces

Recopie du répertoire `Resources` pour qu'il soit vu par l'exécutable généré

(cf. `CompteMots/src/main/CMakeLists.txt` de CompteMots.zip)

```
add_custom_target(copy-resources ALL DEPENDS ${CMAKE_CURRENT_BINARY_DIR}/resources)
file(GLOB RESOURCES CONFIGURE_DEPENDS ${CMAKE_SOURCE_DIR}/resources/*.*)
add_custom_command(OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/resources
        DEPENDS ${CMAKE_SOURCE_DIR}/resources
        COMMAND ${CMAKE_COMMAND} -E make_directory ${CMAKE_CURRENT_BINARY_DIR}/resources
        COMMAND ${CMAKE_COMMAND} -E copy_if_different
        ${RESOURCES}
        ${CMAKE_CURRENT_BINARY_DIR}/resources)
add_dependencies(nomDeVotreExecutable copy-resources)
```

NB : Sous Windows, cette astuce ne fonctionne pas avec un `cmake` exécuté manuellement (cf. section Mini-TP) quand on utilise le générateur par défaut (*Visual Studio 17 2022*) : Lors du `cmake --build`, il y a 2 warnings « Cela peut entraîner un fonctionnement incorrect de la build incrémentielle. », et le répertoire `resources` n'est pas recopié au bon endroit. Pour pallier ce problème, il faut utiliser le générateur *Ninja* en exécutant `cmake -G Ninja -B buildNinja`, suivi de `cmake --build buildNinja` dans un *Powershell* ouvert via *Visual Studio* (Menu "Outils > Ligne de commande > Powershell développeur") ou un *Powershell* simple avec la commande `& 'C:\Program Files\Microsoft Visual Studio\2022\Community\Common7\Tools\Launch-VsDevShell.ps1'`, cf. cette doc).

Ajouter une option pour l'éditeur de liens

Par exemple, pour que l'exécutable puisse être multithreadé sous Unix (cf. `SixQuiPrend/src/test/CMakeLists.txt` de SixQuiPrend.zip)

```
if (UNIX)
    target_link_options(nomDeVotreExecutable PRIVATE -pthread)
endif()
```

Pour mettre le niveau maximum de warnings à la compilation

```
if (MSVC)
    # warning level 4 (see https://docs.microsoft.com/fr-fr/cpp/build/reference/compiler-option-warning-level?view=vs-2019)
    # We do not put /WX to consider all warnings as errors
    add_compile_options(/W4)
else()
    # lots of warnings
    # We do not put -Werror to consider all warnings as errors
    add_compile_options(-g -Wall -Wextra)
endif()
```

Sur les projets écrits exclusivement en C, imposer l'éditeur de liens C++

Certaines entités logicielles (par exemple, *sqlite*) sont écrites complètement en C. Dans ce cas, il faut expressément dire à *CMake* qu'il doit utiliser l'éditeur de lien C++ et pas l'éditeur de lien C :

```
add_library(sqlite sqlite3.c)
set_target_properties(sqlite PROPERTIES LINKER_LANGUAGE CXX)
```

Générer manuellement une version Debug ou une version Release

- Version Debug : `cmake -DCMAKE_BUILD_TYPE=debug -b buildM`
- Version Release : `cmake -DCMAKE_BUILD_TYPE=release -b buildM`

S'appuyer sur des bibliothèques installées manuellement sur la machine

```
set (PHOTON_DIR "C:/Software/Photon-Windows-Sdk_v4-1-16-3")
if(MSVC)
  find_library(PHOTON_COMMON Common-cpp_vc16_debug_windows_md_x64.lib PATHS ${PHOTON_DIR}/Common-cpp/lib REQUIRED)
  find_library(PHOTON_PHOTON Photon-cpp_vc16_debug_windows_md_x64.lib PATHS ${PHOTON_DIR}/Photon-cpp/lib REQUIRED)
  find_library(PHOTON_LOADBALANCING LoadBalancing-cpp_vc16_debug_windows_md_x64.lib PATHS ${PHOTON_DIR}/LoadBalancing-cpp/lib REQUIRED)
elseif(APPLE)
  find_library(PHOTON_COMMON libCommon-cpp_debug_macosx.a PATHS ${PHOTON_DIR}/Common-cpp/lib REQUIRED)
  find_library(PHOTON_PHOTON libPhoton-cpp_debug_macosx.a PATHS ${PHOTON_DIR}/Photon-cpp/lib REQUIRED)
  find_library(PHOTON_LOADBALANCING libLoadBalancing-cpp_debug_macosx.a PATHS ${PHOTON_DIR}/LoadBalancing-cpp/lib REQUIRED)
else()
  # Linux
  find_library(PHOTON_COMMON libCommonDebug64.a PATHS ${PHOTON_DIR}/Common-cpp REQUIRED)
  find_library(PHOTON_PHOTON libPhotonDebug64.a PATHS ${PHOTON_DIR}/Photon-cpp REQUIRED)
  find_library(PHOTON_LOADBALANCING libLoadBalancingDebug64.a PATHS ${PHOTON_DIR}/LoadBalancing-cpp REQUIRED)
endif()
target_link_libraries(nomDeVotreExecutable PUBLIC sfml-graphics ${PHOTON_LOADBALANCING} ${PHOTON_PHOTON} ${PHOTON_COMMON} )
```

Conclusion

1. Objectif de *CMake*
2. Mini-TP
3. Principales notions
4. Trucs et astuces

Pour aller plus loin, ce tutoriel explique notamment comment construire un package d'installation avec l'instruction `cpack`.

Un ancien JIN m'a parlé un jour de « l'enfer *CMake* »… ⇒ *CMake*, un enfer nécessaire ?