LE MULTIPLATEFORME EN C++

A LA RECHERCHE D'UN BUILD TOOL

Clément Doumergue & Roman Sztergbaum

LE MULTIPLATEFORME EN C++

- Un standard pour le langage
- Un standard pour la STL
- Plusieurs implémentations
- Des environnements très différents

FONCTIONNALITÉS REQUISES

- Multiplateforme
- Puissant

FONCTIONNALITÉS OPTIONNELLES

- Facile à appréhender
- Intégré aux autres outils (IDEs, bibliothèques, ...)

LES OUTILS QUE VOUS CONNAISSEZ

GNU MAKE

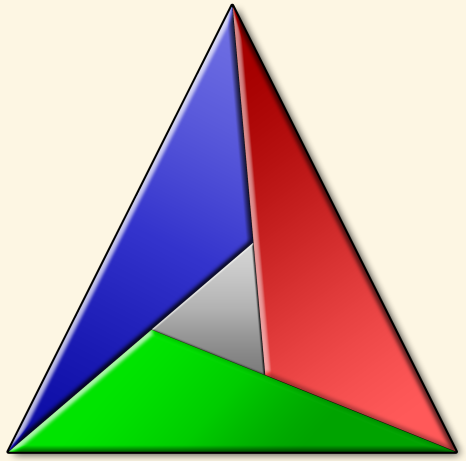
- Simple pour un usage basique
- Très complet
- Trop compliqué pour une utilisation portable

“Writing portable Makefiles is an art.”

MSBUILD

- Configuration en XML
- Pensé pour Windows

PASSER AU NIVEAU SUPÉRIEUR



CMAKE

- Génère des configurations pour les autres build tools
- Scripté via un langage dédié
- Bien supporté par les IDEs

EXEMPLE POUR UN EXÉCUTABLE C++

```
#Le nom de notre projet
project(lazy_proj)

#Définitions de variables
set(CMAKE_CXX_STANDARD 17)
set(lazy_SRC main.cpp)

#Ajout d'un exécutable à build
add_executable(lazy ${lazy_SRC})
```

DÉTECTER LE COMPILATEUR

```
#On choisit les flags appropriés selon le compilateur utilisé
if(MSVC)
    set(CMAKE_CXX_FLAGS "/W4")
else()
    set(CMAKE_CXX_FLAGS "-Wall -Wextra")
endif()
```

CHERCHER DES DÉPENDANCES

AVEC UN SCRIPT : ZLIB

```
#Invocation du script de recherche
find_package(ZLIB)

#Ajout des dossiers d'include trouvés pour une target
target_include_directories(lazy ${ZLIB_INCLUDE_DIR})

#Linkage d'une target sur les bibliothèques trouvées
target_link_libraries(lazy ${ZLIB_LIBRARIES})
```

AVEC UN SCRIPT EXTERNE : SFML

```
#Ajout d'un dossier contenant des scripts de recherche
set(CMAKE_MODULE_PATH ${CMAKE_SOURCE_DIR}/Modules)

#Invocation du script de recherche
find_package(SFML COMPONENTS audio system window graphics)

#Ajout des dossiers d'include trouvés pour une target
target_include_directories(lazy ${SFML_INCLUDE_DIR})

#Linkage d'une target sur les bibliothèques trouvées
target_link_libraries(lazy ${SFML_LIBRARIES})
```

ORGANISER UN PROJET AVEC CMAKE

STRUCTURER UN PROJET

OUT-OF-SOURCE BUILDS

```
> cmake . && ls
CMakeCache.txt CMakeFiles cmake_install.cmake CMakeLists.txt ...
```

```
if(${CMAKE_SOURCE_DIR} STREQUAL ${CMAKE_BINARY_DIR})
    message(FATAL_ERROR "Prevented in-tree built")
endif()
```

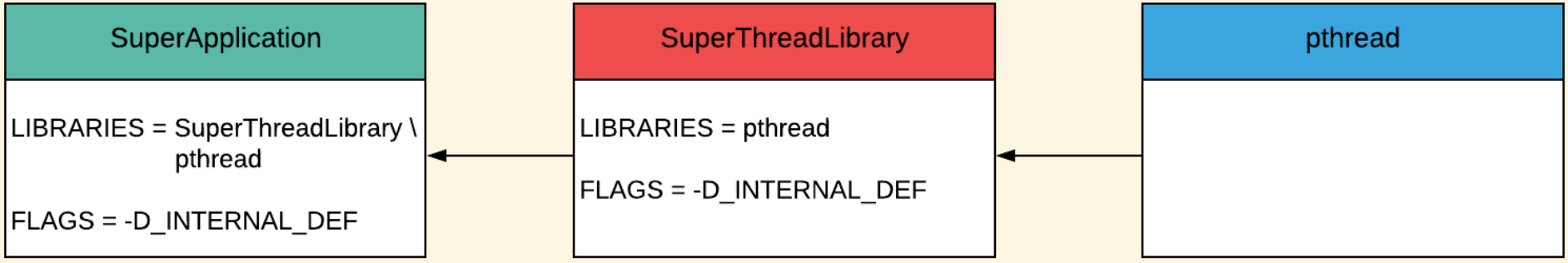
DÉLÉGUER À UN SOUS-CMAKE

```
#Dans le dossier "tests", on a un fichier CMakeLists.txt
add_subdirectory(tests)
```

DÉCOUPER EN MODULES

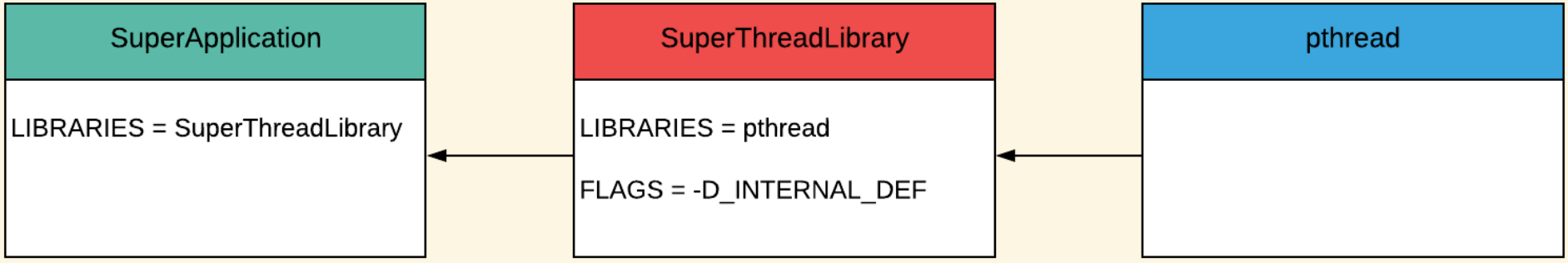
- Réutiliser le code
- Importer seulement ce qui est nécessaire
- Gérer facilement les dépendances
- Traiter les modules comme des bibliothèques

FLAGS VS MODULES



Organisation par flags

FLAGS VS MODULES



Organisation par modules

“TALK IS CHEAP. SHOW ME THE CODE.”

LINUS TORVALDS

LES DÉPENDANCES D'UN MODULE

```
#Notre module a besoin de Lua 5.3
find_package(Lua 5.3)

add_library(scriptlib SHARED ScriptLib.hpp ScriptLib.cpp)

#On a besoin de Lua uniquement à l'intérieur de scriptlib
#Par contre, l'API que l'on expose au client utilise Core::Utils
target_link_libraries(scriptlib PRIVATE ${LUA_LIBRARIES}
                                PUBLIC Core::Utils)

#Idem, le client n'a pas besoin de voir les headers de Lua
target_include_directories(scriptlib PRIVATE ${LUA_INCLUDE_DIR})
```

L'EXEMPLE DE CORE-V2

CRÉER UN MODULE

```
#Inclusion du fichier décrivant les sources du module
#(via la variable MODULE_SOURCES)
include(CMakeSources.cmake)

set(MODULE_PATH ${CMAKE_CURRENT_SOURCE_DIR})
CREATE_MODULE(Core::Meta "${MODULE_SOURCES}" ${MODULE_PATH})

#Link du module sur ses dépendances
target_link_libraries(Core-Meta INTERFACE Core::Config Core::PP)
```

UTILISER LE MODULE

```
#On ajoute le dossier du module à CMake
add_subdirectory(core-v2/core/meta)

#On crée une target
add_executable(meta-demo "${SOURCES_FOR_META_DEMO}")

#On peut linker sur le module comme sur une bibliothèque
target_link_libraries(meta-demo Core::Meta)
```

THAT'S ALL FOLKS !

Slides + Exemples : github.com/squid-lab/multiplatform-cpp-cmake