

* Un projet en C++

Considérons un projet en C/C++, il est formé de plusieurs fichiers header (.h) et de fichiers sources (.c/.cpp) et d'un fichier main.cpp la partie d'entrée du projet, c'est lui qui contient l'unique f^e main du projet!

Un projet C++ peut aussi utiliser des bibliothèques i.e. des fichiers binaires contenant des classes ou f^e standard.

Pour obtenir un exécutable 3 étapes ont lieu :

. Preprocessing: Cette étape est réalisée par le préprocesseur, il va traiter les lignes commençant par # (dites directive).

Il traite les classes très simplement et réalise principalement du opert de "copier - coller" ainsi :

- #include ajoute le contenu d'un fichier dans celui courant.

- #define réalise une substitution de caractères i.e. remplace le symbol par sa valeur

....

À la fin de cette étape on obtient un fichier texte sans directive, elles ont été résolues.

. Compilation: Chaque fichier source sont compilés indépendamment en fichier objet qui sont des fichiers binaires.

- de compilateur traduit le C++ , en langage assembleur puis en code binaire i.e. en langage machine : sera représenté que le CPU peut directement comprendre. Ce fichier en binaire est le fichier dit objet.

- de compilateur affecte à chaque donnée (variable & fonct) une adresse mémoire, là où elle sera stockée lors de l'exéc^tu^t. La correspondance entre le symbol (/name) de la donnée et son adresse mémoire est donné par la table des symboles.

Cependant certaines fonct ou variables peuvent être définies dans d'autre fichier, alors l'adresse est indépnd à cette étape, ça nous à l'éditeur de lier de résoudre la référence.

. Édition de liens: Il prend chaque fichier binaire (objets et ceux des librairies externes) et génère le fichier exécutable : dans de cette étape, l'éditeur de liens essaie de résoudre un max. de références non satisfaites, pour librairies statique.

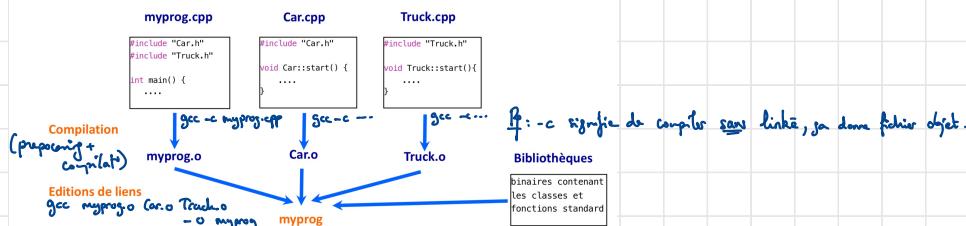
Rg: les binaires des bibliothèques dynamiques ne sont pas ajouté à l'exécutable, mais sont chargées en mémoire au moment de l'exéc^tu^t.

Cela permet de réduire la taille de l'exécutable.

Compilation et édition de liens

• C/C++ : ce processus doit être explicité (e.g. par un [Makefile](#))

• Java : c'est "magique" : fait automatiquement !



Principales options des compilateurs GNU g++ et clang

- version du langage : `-std=c++11`, etc.
- erreurs : `-Wall`
- déboguer : `-g`
- optimisation : `-O1 -O2 -O3 -Os`

Attention : plusieurs versions de C++

=> avoir un compilateur et des bibliothèques à jour !

Rg: On a décrit ici la compilation séparée, elle se caractérise par la compilation indépendante de chaque fichier source (.cpp) ce qui permet de ne pas à avoir à recompiler tout le projet lorsque seulement quelques fichiers sources sont modifiés.

Alors à l'exécution (au runtime), le système d'exploitation charge en mémoire l'exécutables (et aussi ceux des bibliothèques dynamiques). Pour automatiser ce processus on peut utiliser le logiciel CMake. Ce dernier est un multi-système de constructeur logiciel multiplateforme. I.e. il permet de décrire l'organisation des projets e.g. quelles exécutables le projet crée, quelles sont leurs dépendances / les bibliothèques qu'ils utilisent ... Cette description permet à CMake d'autonomement générer des fichiers de configuration pour des systèmes de construction adaptés à chaque plateforme. Cette description est réalisée à l'aide de un ou plusieurs CMakeLists.txt.

CMake en générant ce fichier déclenche la construction du projet (compilation + édition de liens) à un autre logiciel physique = la plateforme (e.g. sur Windows génère une solution Visual Studio, sur Linux génère make file...).

P: Sur windows pour accéder à CMake en ligne de commande on peut :

- Ouvrir Terminal puis Développeur PowerShell for VS 2022 alors c'est connu.

- ou - — Visual Studio puis Outils > ligne de Commande > PowerShell développeur.

Si - dans ce trouve différents CMake files types pour différents projets de tailles différentes.

- Un petit projet : Imaginons un petit projet formé de quelques fichiers sources et qui crée un exécutables:

Organisé : - smallProject /

- ↳ main.cpp

- ↳ src1.h (e.g. point.h)

- ↳ src1.cpp (e.g. Point.cpp)

- ↳ CMakeLists.txt

- ↳ build/ (pour gérer le build, doit être mis dans gitignore)

Alors CMakeLists.txt est chargé de générer l'exécutables, ainsi c'est le fichier CMake en racine du projet il doit donc :

- Premièrement, en 1^{re} ligne, la version minimal de CMake requise pour construire le projet :

`cmake_minimum_required(VERSION <min>)` avec `<min>` le numéro de version e.g. 3.28 .

- Déclarer le projet par `project(<nomDuProjet>)`

[VERSION <version>]

[DESCRIPTION <desc>]

[LANGUAGES <lang>])

avec [...] indique paramètre optionnel. Pour projet en C++ <lang> est CXX.

- Déclarer une target représentant l'exécutables et définir les fichiers sources depuis il dépend (peut aussi lister headers pour l'IDE) : `add_executable(<name> <source>...<source>)` le nom de l'exécutables est `<name>` et il dépend de `<source>`..., ainsi `<source>` est main.cpp.



Un moyen projet: Si un projet prend de l'ampleur il est bon de l'organiser en différents domaines. Un particulier on peut regrouper les fichiers sources en module qui peuvent être une comme une bibliothèque interne au projet. Pour chaque fichier de sa bibliothèque on sépare la fichier source .cpp dans un dossier src et la header dans un dossier include. Puis on sépare le CMakeLists.txt en différents morceaux/fichiers pour rendre le projet plus modulaire en donnant plus d'autonomie à ses composantes.

Organisé : - mediumProject /

 └ src /

 └ module1 /

 └ src1.cpp ...

 └ CMakeLists.txt

 └ module2 /

 └ src2.cpp ...

 └ CMakeLists.txt

fichiers source des modules
bibliothèques internes (peut aussi avoir header "privé"
non accessible au public).

Chaque module = son CMake

qui la gère

Délégue la gérance des modules.

└ CMakeLists.txt

└ include /

 └ mediumProject /

 └ src1.h

 └ src2.h

] class include/mediumProject se trouve la headers publiques de notre bibliothèques / nos modules.

¶: 2 domaines indiqués pour des raisons de namespace.

Alors - mediumProject / CMakeLists.txt: il se charge de gérer l'environnement, ainsi / au small project :

- pour déléguer la création des bibliothèques il appelle add_subdirectory (src) qui va "exécuter" le CMakeLists.txt trouvant dans le dossier.
- pour linker, indiquer la dépendance d'un target (eg. l'environnement) aux bibliothèques, on utilise la commande target_link_libraries (<target> PRIVATE <item>)

PUBLIC

INTERFACE

avec <item> qui peut être : - une library target nomée créée par add_library() (cf ci-dessous)
c'est le cas avec bibliothèque interne par exemple.
- le chemin absolu vers les binaries d'une bibliothèque (correspond à des fichiers objets) eg. .lib sur Windows.
- le nom d'une bibliothèque alors CMake demandera au linker de chercher la bibliothèque.
...

. PRIVATE : si veut cacher la bibliothèque de l'utilisateur (?)

PUBLIC : tout le monde peut la voir.

¶: On peut link une bibliothèque à une autre bibliothèque.



- `mediumProject/src/CMakeLists.txt`: ne fait pas de liaison aux différents modules donc n'appelle que `add-library()`

- `mediumProject/src/module1/CMakeLists.txt`: est chargé de générer la bibliothèque / module 1, ainsi:

- pour avoir dans l'IDE les headers de la bibliothèque on peut définir une variable qui les liste:

```
set(HEADER_LIST "## {CMAKE_SOURCE_DIR}/include/mediumProject/module1.h")  
    nom de la variable  
    variable définit par CMake  
    qui contient toutes  
    les headers du projet
```

- définit la target de la bibliothèque par `add-library()`. On distingue:

- `add-library (<name> [STATIC | SHARED] <source 1> <source 2...>)`: ajoute une bibliothèque `<name>` au build qui est composée des fichiers `<source 1>...`

`STATIC` signifie que la bibliothèque générée sera composée d'object files à être linké à des targets.

`SHARED` signifie que la bibliothèque sera dynamique.

- `add-library (<name> INTERFACE)` permet de déclarer une header only bibliothèque i.e. une bibliothèque qui n'a pas besoin d'être compilé en fichiers objets, par conséquent pas de fichiers sources. P: Pour indiquer où se trouvent headers on utilise `target_include_directories()` cf ci-dessous.

- précise où se trouve les fichiers include i.e. headers publics de la target bibliothèque par `target_include_directories(<target> [<item>])`

`<item>` est en général l'included dossier i.e. "root/include".

`PUBLIC` signifie que tout autre target link sur module aura aussi besoin du dossier include

`PRIVATE` signifie que seul le module = besoin des includes.

P: Pour inclure un module / bibliothèque définit la sorte ce n'est pas fininclude "path/module.h"

MAIS: #include <mediumProject/module.h>

Un gros projet: On considère un projet similaire au précédent mais en plus il dépend de third party library, comme ces fichiers ne sont pas à modifier, ils sont "externe" au projet. Pour cette raison on ajoute un dossier extern au projet.

P: On peut aussi appeler ces third party librairie des external dependencies du projet.

On propose différentes approches pour ajouter ces dépendances au projet:

- Télécharger les binaries de la librairie et les ajouter au dossier extern de notre projet, alors on peut la link au projet avec `target_link_libraries()` avant).

!! Cela nécessite d'avoir plusieurs binaries en fonction du système d'exploitation... C'est pour ça qu'on écrit cette approche...



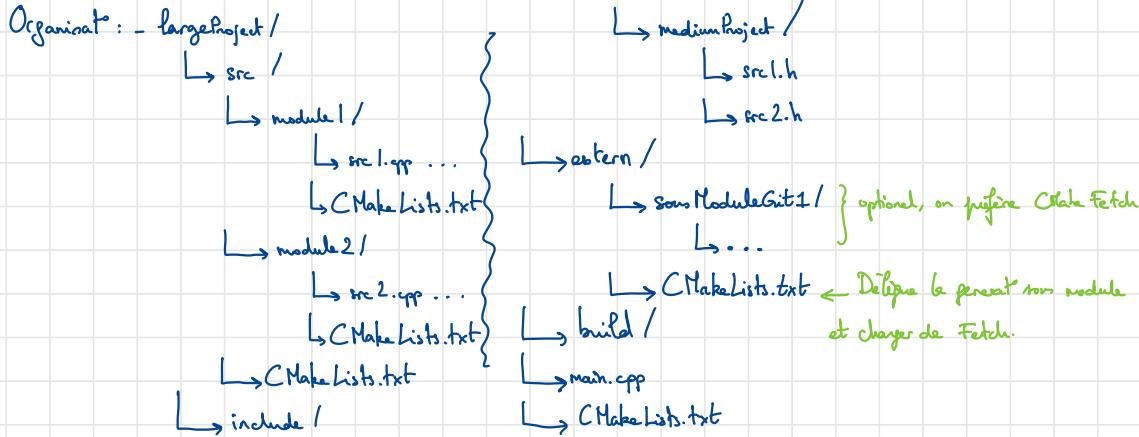
- Télécharger les fichiers sources, de préférence comme sous-module git. Pour cela on ajoute le repo au dossier externe par "git submodule add git@.... extern". Une fois fichier source ajouté il suffit de l'inclure au projet avec add_subdirectory si sous module supporte CMake : Le CMake du sous-module crée une target qui on pourra utiliser avec target_link_libraries.

!! Cela demande à une personne qui cloneait le projet d'utiliser l'opt --recursive lors du clonage du repo projet, afin de cloner aussi les sous-modules...

- Utilise CMake FetchContent, cette commande CMake permet d'ajouter au projet du contenu ne trouvant en ligne (à partir URL, repo git...). Si ce contenu supporte CMake on pourra y faire référence par la target qu'il crée. Cette approche est expliquée plus loin.

R: Cette dernière méthode est à favoriser

. Cette méthode n'ajoute pas le contenu directement au projet, mais l'ajoute par un dossier _deps créé par CMake dans dossier build.



Alors les fichiers CMake sont très similaires à ceux du mediumProject, cependant :

- LargeProject/CMakeLists.txt : il se charge de générer l'exécutable, ainsi il est très similaire au medium project. Il ne faut juste pas oublier de déléguer la génération des dépendances externes et de link explicitement cette dépendance à l'exécutable. (Règle : close pour library interne au projet).

- LargeProject/extern/CMakeLists.txt : il est chargé de déléguer la génération de sous module au parent, et aussi de fetch contenu. Fetch du contenu se fait en 3 étapes :



- `include(FetchContent)` : pour indiquer à CMake qu'on va utiliser la fonctionnalité FetchContent.

- `FetchContent_Declare(`

`<nomDuContenu>`

`GIT_REPOSITORY <url du repo git>`

`GIT_TAG <git-tag>`

`)` : permet de déclarer son `<nomDuContenu>` ce que doit fetch CMake.

- `FetchContent_MakeAvailable(<nomDuContenu>)` rend disponible le contenu (va le "télécharger" et appeler `add_subdirectory` pour désigner sa racine). Si contenu supporte CMake on dispose alors d'une target pour désigner ce contenu.