Introduction au moteur de production CMake

December 5, 2022

GNU toolchain

Les étapes de compilation

Fichier source (`main.c`)

```c
#define VALUE 5

int main() {
  return VALUE;
}
```

Les étapes de compilation

Après l'étape de *preprocessing* (`main.i`)

```
1  int main() {
2    return 5;
3  }
```

Les étapes de compilation

Après compilation (`main.S`)

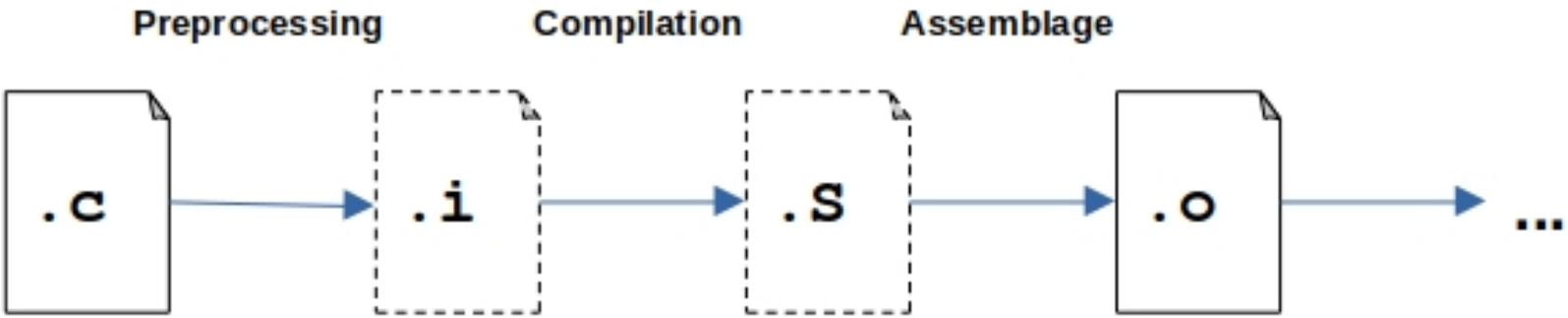
```
 5  main:
 6  .LFB0:
 7          .cfi_startproc
 8          endbr64
 9          pushq   %rbp
10          .cfi_def_cfa_offset 16
11          .cfi_offset 6, -16
12          movq    %rsp, %rbp
13          .cfi_def_cfa_register 6
14          movl    $5, %eax
15          popq    %rbp
16          .cfi_def_cfa 7, 8
17          ret
18          .cfi_endproc
```

Les étapes de compilation

Après assemblage (`main.o`)

```
1
2  main.o:     file format elf64-x86-64
3
4
5  Disassembly of section .text:
6
7  0000000000000000 <main>:
8     0:   f3 0f 1e fa                endbr64
9     4:   55                         push   %rbp
10    5:   48 89 e5                   mov    %rsp,%rbp
11    8:   b8 05 00 00 00             mov    $0x5,%eax
12    d:   5d                         pop    %rbp
13    e:   c3                         ret
```

Les étapes de compilation



Fonctionnement des entêtes

```
project/
├── main.c
└── aux.c
```

main.c

```c
#include <stdio.h>

int main() {
  printf("My value = %d\n", my_function());
  return 0;
}
```

aux.c

```c
#include <stdio.h>

void my_function(void) {
  printf("hello\n");
}
```

Fonctionnement des entêtes

```
$ gcc aux.c main_for_aux.c
main_for_aux.c: In function ‘main’:
main_for_aux.c:4:29: warning: implicit declaration of
 ↪  function ‘my_function’
 ↪  [-Wimplicit-function-declaration]
    4 |   printf("My value = %d\n", my_function());
      |                             ^~~~~~~~~~~
$ ./a.out
hello
My value = 6
```

Fonctionnement des entêtes

```
project/
├── main.c
├── aux.c
└── aux.h
```

main.c

```c
#include <stdio.h>

#include "aux.h"

int main() {
  printf("My value = %d\n", my_function());
  return 0;
}
```

aux.c

```c
#include <stdio.h>

#include "aux.h"

void my_function(void) {
  printf("hello\n");
}
```

aux.h

```c
#pragma once

void my_function(void);
```

Fonctionnement des entêtes

```
$ gcc aux.c main_for_aux.c
main_for_aux.c: In function ‘main’:
main_for_aux.c:6:29: error: invalid use of void
 ↪  expression
    6 |   printf("My value = %d\n", my_function());
      |
```

Fonctionnement des entêtes

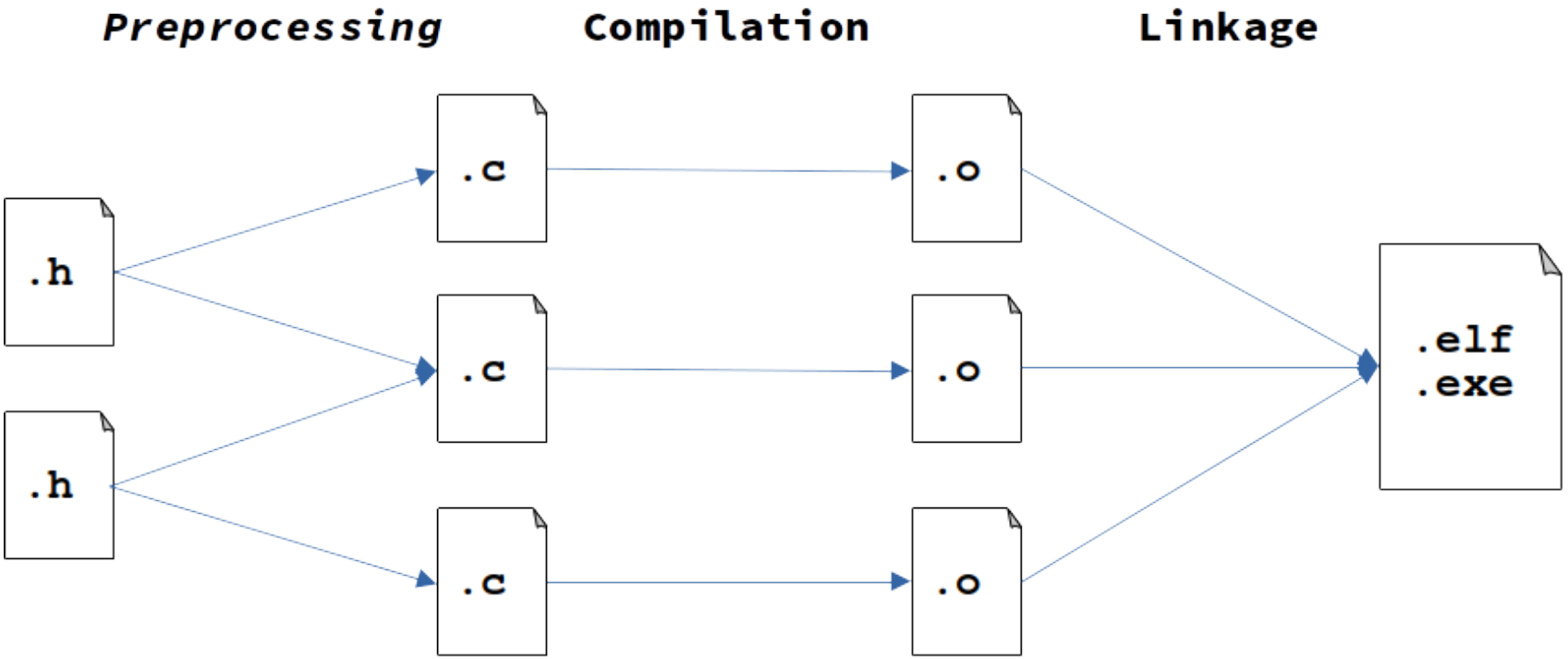
Avec entêtes

- **Traditionnellement**, les entêtes contiennent les **déclarations** des fonctions
- Souvent appairées avec des fichiers sources, qui contiennent les **définitions**
- Besoin de synchroniser l'entête avec son fichier source
- Système utilisé par les langages relativement vieux (C, C++, ...)

Sans entêtes

- Les fichiers sources contiennent les **définitions** des fonctions et peuvent être "inclus" directement.
- Les fichiers sources se suffisent à eux-mêmes
- Système utilisé par les langages récents (Python, Java, Rust, C++, ...)

Linkage



GNU Make

Principe

Structure d'une règle

```
1 cible: dependances...
2     recette...
```

Exemple

```
1 application: main.o aux.o
2     gcc main.o aux.o
```

Principe

```
project/
├── main.c
├── aux.c
├── aux.h
└── Makefile
```

Makefile

```
1  application: main.o aux.o
2          gcc main.o aux.o
```

(Règle pour générer `main.o` ?)
(Règle pour générer `aux.o` ?)

Principe

```
project/
├── main.c
├── aux.c
├── aux.h
└── Makefile
```

Makefile

```
1  application: main.o aux.o
2          gcc main.o aux.o
3
4  main.o: main.c
5          gcc -c main.c -o main.o
```

(Règle pour générer `aux.o` ?)

Principe

```
project/
├── main.c
├── aux.c
├── aux.h
└── Makefile
```

Makefile

```
1  application: main.o aux.o
2          gcc main.o aux.o
3
4  main.o: main.c
5          gcc -c main.c -o main.o
6
7  aux.o: aux.c
8          gcc -c aux.c -o aux.o
```

Relation avec CMake

GNU Make

- *Build system*
- Permet de produire des **binaires** (exécutables, bibliothèques statiques et dynamiques...)

CMake

- Générateur de *build systems*
- Permet de générer des **Makefiles**

CMake s'utilise donc avec Makefile (ou un autre build system)

CMake

Le *source tree* et le *binary tree*

Source tree

```
project/
├── CMakeLists.txt
├── main.c
├── src/
│   ├── CMakeLists.txt
│   ├── aux_a.c
│   └── aux_b.c
└── include/
    ├── CMakeLists.txt
    ├── aux_a.h
    └── aux_b.h
```

Binary tree ?

Le *source tree* et le *binary tree*

Source tree

```
project/
├── CMakeLists.txt
├── main.c
├── src/
│   ├── CMakeLists.txt
│   ├── aux_a.c
│   └── aux_b.c
└── include/
    ├── CMakeLists.txt
    ├── aux_a.h
    └── aux_b.h
```

Binary tree

```
build/
├── Makefile
├── MyExecutable
├── src/
│   ├── Makefile
│   └── libMyLibrary.a
└── include/
    └── Makefile
```

Note: en réalité, le *binary tree* comporte plus de fichiers, mais par souci de clarté, seuls les fichiers principaux ont été gardé.

Le *source tree* et le *binary tree*

```
project/
├── CMakeLists.txt
├── main.c
├── build/ ...  En général, le binary tree se trouve dans
│               un dossier nommé build/ situé à la racine
│               du projet
│   ├── Makefile
│   ├── MyExecutable
│   ├── src/
│   │   ├── Makefile
│   │   └── libMyLibrary.a ...  Bibliothèque statique
│   └── include/
│       └── Makefile
├── src/
│   ├── CMakeLists.txt
│   ├── aux_a.c
│   └── aux_b.c
└── include/
    ├── CMakeLists.txt
    ├── aux_a.h
    └── aux_b.h
```

L'étape de configuration et de génération

Commande:

```
$ cmake -B build
```

Résultat ?

Source tree

```
project/
├── CMakeLists.txt
├── main.c
├── src/
│   ├── CMakeLists.txt
│   ├── aux_a.c
│   └── aux_b.c
└── include/
    ├── CMakeLists.txt
    ├── aux_a.h
    └── aux_b.h
```

Binary tree

```
build/
├── ????????
├── src/
│   └── ????????
└── include/
    └── ????????
```

L'étape de configuration et de génération

Commande:

```
$ cmake -B build
```

Résultat:

Source tree

```
project/
├── CMakeLists.txt
├── main.c
├── src/
│   ├── CMakeLists.txt
│   ├── aux_a.c
│   └── aux_b.c
└── include/
    ├── CMakeLists.txt
    ├── aux_a.h
    └── aux_b.h
```

Binary tree

```
build/
├── Makefile
├── src/
│   └── Makefile
└── include/
    └── Makefile
```

L'étape de construction du projet

Commande:

`$ cmake --build build OU $ make -C build`

Résultat ?

Source tree

```
project/
├── CMakeLists.txt
├── main.c
├── src/
│   ├── CMakeLists.txt
│   ├── aux_a.c
│   └── aux_b.c
└── include/
    ├── CMakeLists.txt
    ├── aux_a.h
    └── aux_b.h
```

Binary tree

```
build/
├── Makefile
├── ??????????
├── src/
│   ├── Makefile
│   └── ????????????
└── include/
    └── Makefile
```

L'étape de construction du projet

Commande:

```
$ cmake --build build OU $ make -C build
```

Résultat:

Source tree

```
project/
├── CMakeLists.txt
├── main.c
├── src/
│   ├── CMakeLists.txt
│   ├── aux_a.c
│   └── aux_b.c
└── include/
    ├── CMakeLists.txt
    ├── aux_a.h
    └── aux_b.h
```

Binary tree

```
build/
├── Makefile
├── MyExecutable
├── src/
│   ├── Makefile
│   └── libMyLibrary.a
└── include/
    └── Makefile
```

Commandes pour le TP

add_executable

Contenu de `CMakeLists.txt`:

```
add_executable(MyExecutable main.c aux.c)
```

Résultat:

Source tree

```
project/
├── CMakeLists.txt
├── main.c
└── aux.c
```

Binary tree

```
build/
├── Makefile
└── MyExecutable
```

add_library

Contenu de `CMakeLists.txt`:

```
add_library(MyLibrary aux_a.c aux_b.c aux_c.c)
```

Résultat:

Source tree

```
project/
├── CMakeLists.txt
├── aux_a.c
├── aux_b.c
└── aux_c.c
```

Binary tree

```
build/
├── Makefile
└── libMyLibrary.a
```

target_include_directories

Contenu de `include/CMakeLists.txt`:

```
target_include_directories(MyLibrary PUBLIC ${CMAKE_CURRENT_SOURCE_DIR})
```

(Hypothèse : `src/CMakeLists.txt` définit la bibliothèque `MyLibrary`)

Résultat:

Source tree

```
project/
├── CMakeLists.txt
├── src/
│   ├── CMakeLists.txt
│   ├── aux_a.c
│   ├── aux_b.c
│   └── aux_c.c
└── include/
    ├── CMakeLists.txt
    ├── aux_a.h
    ├── aux_b.h
    └── aux_c.h
```

Aucun résultat visible dans le *build tree*... Cela permet juste d'indiquer au compilateur où se trouvent les entêtes associées à la bibliothèque `MyLibrary`. `${CMAKE_CURRENT_SOURCE_DIR}` est égale au chemin du dossier qui contient le `CMakeLists.txt` en cours d'évaluation.

target_link_libraries

Contenu de CMakeLists.txt:

```
add_executable(MyExecutable main.c)
target_link_libraries(MyExecutable PUBLIC MyLibrary)
```

(Hypothèses : src/CMakeLists.txt définit la bibliothèque MyLibrary et include/CMakeLists.txt déclare include/ comme chemin d'inclusion pour MyLibrary)

Résultat:

Source tree

```
project/
├── CMakeLists.txt
├── main.c
├── src/
│   ├── CMakeLists.txt
│   ├── aux_a.c
│   ├── aux_b.c
│   └── aux_c.c
└── include/
    ├── CMakeLists.txt
    ├── aux_a.h
    ├── aux_b.h
    └── aux_c.h
```

Aucun résultat visible... Cette commande permet d'indiquer au compilateur de linker notre exécutable avec notre bibliothèque, mais aussi d'ajouter les chemins d'inclusions de MyLibrary à MyExecutable.

add_subdirectory

Contenu de `CMakeLists.txt`:

```
add_subdirectory(src)
add_subdirectory(include)
```

Résultat:

Source tree

```
project/
├── CMakeLists.txt
├── src/
│   └── CMakeLists.txt
└── include/
    └── CMakeLists.txt
```

Binary tree

```
project/
├── Makefile
├── src/
│   └── Makefile
└── include/
    └── Makefile
```

TP: Projet CMake basique

TP: Projet CMake basique

Clonez le dépot :

`git clone https://github.com/Club-INTech/Formation-CMake`

et **écrivez un ou plusieurs fichier(s)** `CMakeLists.txt` **afin de pouvoir construire le projet**.
Il y a plusieurs solutions possibles, mais le but de l'exercice est d'y parvenir **sans modifier les fichiers sources**.
Pour vérifier votre solution, lancez l'exécutable obtenu.

Déclarer une dépendance externe avec FetchContent_Declare

Contenu de `CMakeLists.txt`:

```
include(FetchContent)
FetchContent_Declare(
    ExternalDependency
    GIT_REPOSITORY https://github.com/boostorg/boost)
```

Résultat:

Source tree

```
project/
└── CMakeLists.txt
```

Aucun résultat visible... Cette ligne permet de déclarer et de décrire une dépendance via FetchContent (ici, un projet CMake hébergé sur GitHub). Cependant, **la dépendance n'est pas téléchargée** tant qu'elle n'est pas requise.

Solliciter une dépendance externe pour l'utiliser avec FetchContent_MakeAvailable

Contenu de CMakeLists.txt:

```
include(FetchContent)
FetchContent_Declare(
    ExtLib
    GIT_REPOSITORY https://github.com/boostorg/boost)
FetchContent_MakeAvailable(ExtLib)
```

Résultat:

Source tree

```
project/
└── CMakeLists.txt
```

Binary tree

```
build/
└── _deps/
    ├── extlib-build/
    ├── extlib-src/
    └── extlib-subbuild/
```

Exercice rapide

Si vous n'avez pas trouvé de solution à l'exercice précédent, vous pouvez en télécharger une depuis le dépôt.

```
git reset HEAD --hard # Efface toutes vos modifications
git checkout ex2      # Avance jusqu'à l'exercice 2
```

Ensuite, modifier les `CMakeLists.txt` afin de pouvoir indiquer à FetchContent de télécharger la branche `external` du dépôt, et linker la bibliothèque ainsi téléchargée à votre exécutable.

```
FetchContent_Declare(
    <nom pour désigner la dépendance>
    GIT_REPOSITORY <lien vers le dépôt git>
    GIT_TAG <nom de la branche à télécharger>
)
```

Et après ?

- `add_custom_target` et `add_custom_command` pour créer vos propres cibles et règles
- `set` et `list` pour manipuler les variables
- Les fonctions, les structures `if` et `for`
- `include`, `CMAKE_MODULE_PATH` pour créer et utiliser des modules CMake