

Introduction à CMake et C++

Benoit DESROCHERS

ENSTA Bretagne

Fevrier 2016

V1.00

Supports de cours disponibles sur . . .

Section 1

Introduction

UV 4.7 Outils logiciels pour la robotique

Objectif de l'UV

Définir et structurer une architecture logicielle appliquée à la robotique.

Middleware

Un *middleware* (anglicisme) ou *intergiciel* est un logiciel tiers qui crée un réseau d'échange d'informations entre différentes applications informatiques. Il doit :

- ▶ Faciliter le développement/réutilisation des composants logiciels
- ▶ Définir les interfaces et comportement des sous modules
- ▶ Fournir une suite d'outils (affichage, analyse de logs,...)
- ▶ Être robuste et "sans bugs"

UV 4.7 Outils logiciels pour la robotique

Exemples de middleware

ROS, MOOS, Pocolibs, yarp, HLA, zmq, mavlink, DDS, ...

Organisation

- ▶ Introduction C++ et CMake
- ▶ MOOS (2 séances)
- ▶ ROS (3 scéances)
- ▶ Simulation MORSE (2 séances)

Introduction à la compilation et à CMake

Objectifs

Être capable d'organiser / compiler et déployer un projet C++

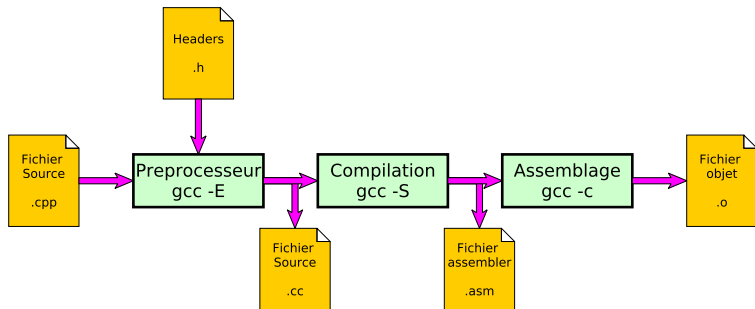
Points abordés

- ▶ Déroulement du processus de compilation
- ▶ Introduction à CMake
- ▶ TD de mise en application

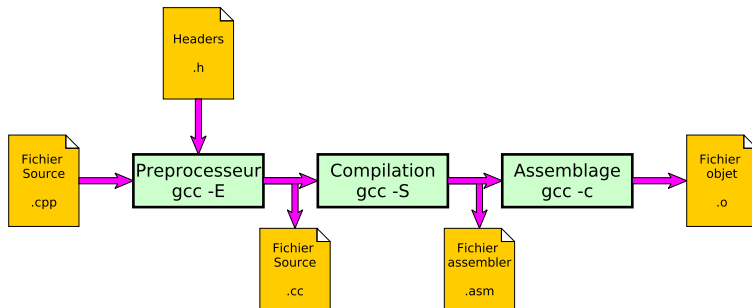
Section 2

Compilation

Étapes de la compilation



Étapes de la compilation



Commandes pour obtenir les fichiers intermédiaires

```
> # Execution du preprocesseur
> gcc -E main.cpp -o main.cc
> # Compilation
> gcc -S main.cpp -o main.
> # Assemblage
> gcc -c main.cpp -o main.o
> # Edition des liens
> gcc -c main.o -o main
```


Exemple 1

main.cpp

```
1  int foo(){  
2      return 3;  
3  }  
4  
5  int main(){  
6      return foo();  
7  }
```

Compilation / Exécution en ligne de commande

```
> gcc main.cpp -o main  
> # execution  
> ./main  
> # affichage de la valeur de retour  
> echo $?  
> 3
```

Exemple 1

Code assembleur *main.s*

```

1  .file "main.cpp"
2  .text
3  .globl _Z3foov
4  .type _Z3foov, @function
5  _Z3foov:
6  .LFB0:
7  .cfi_startproc
8  pushq %rbp
9  .cfi_def_cfa_offset 16
10 .cfi_offset 6, -16
11 movq %rsp, %rbp
12 .cfi_def_cfa_register 6
13 movl $3, %eax
14 popq %rbp
15 .cfi_def_cfa 7, 8
16 ret
17 .cfi_endproc
18 .LFE0:
19 .size _Z3foov, .-_Z3foov

20 .globl main
21 .type main, @function
22 main:
23 .LFB1:
24 .cfi_startproc
25 pushq %rbp
26 .cfi_def_cfa_offset 16
27 .cfi_offset 6, -16
28 movq %rsp, %rbp
29 .cfi_def_cfa_register 6
30 call _Z3foov
31 popq %rbp
32 .cfi_def_cfa 7, 8
33 ret
34 .cfi_endproc
35 .LFE1:
36 .size main, .-main
37 .ident "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.1) 4.8.4"
38 .section .note.GNU-stack,"",@progbits

```

Affichage des symboles contenus dans *main.o*

```

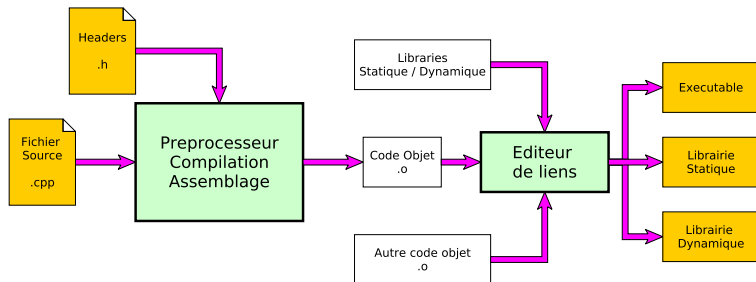
> nm -gC main.o
0000000000000000b T main
0000000000000000 T foo()

```

Édition des liens

Problématique

Lorsque le code est réparti sur plusieurs fichiers *objet* et/ou librairies il faut pouvoir faire les liens entre les symboles.



Exemple 2

foo.h

```
1  // Definition du prototype de la fonction <foo>
2  int foo(int a);
```

foo.cpp

```
1  #include "foo.h"
2
3  int foo(int a){ return a+1; }
```

main.cpp

```
1  #include "foo.h"
2
3  int main(){
4      return foo(4);
5  }
```

Code assembleur généré

main.s

```
.file "main.cpp"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $4, %edi
call _Z3fooi
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.1) 4.8.4"
.section .note.GNU-stack,"",@progbits
```

Code assembleur généré

main.s

```
.file "main.cpp"
.text
.globl main
.type main, @function
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $4, %edi
call _Z3fooi
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size main, .-main
.ident "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.1) 4.8.4"
.section .note.GNU-stack,"",@progbits
```

foo.s

```
.file "foo.cpp"
.text
.globl _Z3fooi
.type _Z3fooi, @function
_Z3fooi:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl %edi, -4(%rbp)
movl -4(%rbp), %eax
addl $1, %eax
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size _Z3fooi, .-_Z3fooi
.ident "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.1) 4.8.4"
.section .note.GNU-stack,"",@progbits
```

Résolution des liens

Contenu des fichiers objets

```
> nm -gC main.o
> 0000000000000000 T main
>                               U foo(int)
> nm -gC foo.o
> 0000000000000000 T foo(int)
> nm -gC main_exe
```

Génération de l'exécutable

Il faut indiquer explicitement où trouver les symboles manquant.

```
> gcc main.o foo.o -o main_exe
```

Librairies Statiques / Dynamiques

Les bibliothèques permettent de ne pas recompilier le code à chaque fois. Il existe deux types de bibliothèques

Librairies Statiques

Le code utilisé est copié dans le binaire à l'édition des liens.

```
> # Creation d'une bibliothèque statique  
> ar rcs libfoo.a foo.o
```

Librairies dynamiques

Les symboles sont résolus lors de l'exécution du programme. La bibliothèque peut être partagée entre plusieurs exécutables.

```
> # Creation d'une bibliothèque dynamique  
> gcc -shared foo.o -o libfoo.so
```


Librairies Statiques / Dynamiques

Commande pour lier la librairie

```
> # linkage (ordre des parametres est important)
> gcc main.o -L. -lfoo -o main
```

Section 3

CMake

CMAKE - Générateur de projet

Outil de génération automatique de projet **libre** et **multiplateforme**.

Il permet de¹ :

- ▶ gérer l'ordre de compilation des fichiers sources et leurs dépendances
- ▶ générer des fichiers de configuration en fonction du système
- ▶ Installer et tester

1. Pour un tutoriel complet visiter <http://www.kitware.com/products/books.php>

Commandes de Base

Ossature minimale d'un fichier CMakeLists.txt

```
cmake_minimum_required (VERSION 2.6)
# Ceci est une commantaire
project (myProjectName)
```

Création des binaires

```
# Creation de l'executable NOM_EXECUTABLE avec
# les fichiers sources src1.cpp src2.cpp, ...
add_executable( NOM_EXECUTABLE src1.cpp src2.cpp)
```

```
# Creation d'une librairie
add_library(NOM_LIB STATIC mylib.cpp )

add_target_libraries(NOM_EXECUTABLE lib1 )
```

Autres commandes utiles

Ajout d'une librairie externe

```
#Ajout des chemins pour les headers
INCLUDE_DIRECTORIES(path/to/include/dir)
# ajout d'une library externe detectable par cmake
find_package(myExternalLib REQUIRED)
INCLUDE_DIRECTORIES(${myExternalLib_INCLUDES})
...
target_link_library(myTarget ${myExternalLib_LIBRARIES} )
```

```
# Ajout d'un sous-repertoire
add_subdirectory(SUBDIR)
```

```
# Installation des binaires
install(TARGETS myExe DESTINATION bin)
install(TARGETS myLib DESTINATION lib)
install(FILES myheader DESTINATION include)
```

Utilisation

Organisation

- ▶ myProject/
 - ▶ CMakeLists.txt
 - ▶ foo.cpp
 - ▶ foo.h
 - ▶ main.cpp

CMakeLists.txt

```
cmake_minimum_required (VERSION 2.6)
project (myProject)
add_executable(main foo.cpp main.cpp)
install(TARGETS main DESTINATION bin)
```

Construction du projet

```
> mkdir build && cd build
> cmake -DCMAKE_INSTALL_PREFIX=/tmp -CMAKE_BUILD_TYPE=Debug ../
> make && make install
```

Questions

Questions ?