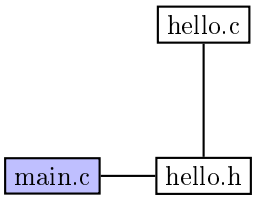
Guide de survie pour Makefile

Un Makefile c'est quoi ?

Les Makefiles sont des fichiers permettant d'executer un ensemble d'actions, comme la compilation d'un projet, la mise à jour d'un rapport, où l'archivage de données. Les Makefiles sont des fichiers shell particulier ils respectent cependant les conventions de codages du shell. L'idée de ce document est de présenté l'idée générale d'un Makefile via un exemple basique, puis d'aller vers un exemple plus complexe, le Makefile pour notre robot.

Un exemple basique

Considérons un exemple de projet très basique, constitué de 3 fichiers : hello.c, hello.h et main.c. Nous pouvons faire le graphe des dépendances suivants



De manière plus littérale nous avons définit des fonctions dans hello.c, ces fonctions sont referencées par leurs prototypes dans hello.h. Par ailleurs nous avons définies des fonctions dans main.c, nous sous entendons que les fonctions de main.c font appels à celles de hello.c (d'où l'inclusion de hello.h dans main.c).

Nous voulons donc compiler main.c pour créer un executable que nous appellerons *project*. Si nous essayons directement de faire "gcc -Wall -Werror -std=c99 main.c project" nous allons avoir des références indéfinies. En effet il faut d'abbord compiler hello.c en hello.o pour que la machine connaisse les fonctions définit dans le hello.c (c'est le rôle du hello.o). Ainsi pour créer project il faut faire deux lignes de commande "gcc -Wall -Werror -std=c99 hello.c -c" puis "gcc -Wall -Werror -std=c99 hello.o main.c -o project".

Deux nouvelles options se présentent "-c" qui permet de dire au compilateur de créer un fichier ".o" et "-o" qui permet d'appeler le linker (c'est lui qui dit au compilateur que certainnes fonctions appelées dans main.c sont définies dans hello.o).

Maintenant que nous avons compris comment nous pouvions compiler ce projet, nous allons regarder comment automatiser ces étapes. Pour cela regardons la synthaxe type d'un Makefile.

```
cible : dependances
        commandes
```

Analysons ces deux lignes de commandes, le “cible” permet de donner un nom ainsi dans le shell nous pourrons taper “make cible” pour effectuer les actions correspondantes. Dans notre cas la cible serait “project”. Ensuite les “dépendances” sont les fichiers nécessaires à la création de la cible, dans notre cas cela serait “hello.o”. Un exemple de Makefile complet pour ce projet serait le suivant

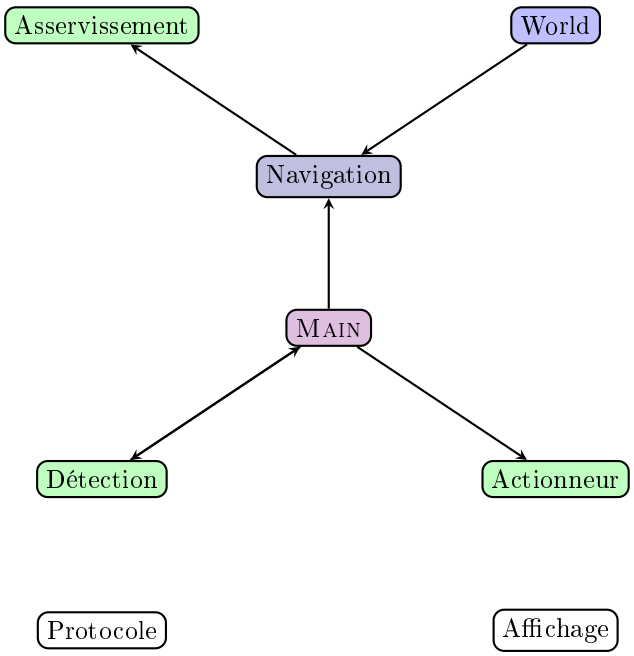
```
hello.o:
        gcc -Wall -Werror -std=c99 hello.c -c

project : hello.o
          gcc -Wall -Werror -std=c99 hello.o main.c -o project
```

Un exemple plus complexe

Nous allons maintenant passer à un exemple plus complexe, l’exemple du Makefile utilisé pour le robot. Plusieurs complications et exigences vont se révéler avec ce projet

1. Beaucoup de fichiers pour le projet
2. Des dépendances entre les fichiers
3. Arreter de compiler les dépendances “à la main”
4. Avoir deux main (un main et un test)
5. Nettoyer son dossier
6. Vouloir changer les options de compilation / le compilateur facilement
7. Avoir des fichiers dans différents dossier

Commençons par présenter les différents fichiers du projet.




```
code/
| Makefile
| rasp
|    | src
|    |    | actionneur.hpp
|    |    | affichage.cpp
|    |    | affichage.hpp
|    |    | asserv.cpp
|    |    | asserv.hpp
|    |    | detection.cpp
|    |    | detection.hpp
|    |    | main.cpp
|    |    | main.hpp
|    |    | navigation.cpp
|    |    | navigation.hpp
|    |    | protocole.cpp
|    |    | protocole.hpp
|    |    | world.cpp
|    |    | world.hpp
|    | tst
|    |    | test.cpp
|    |    | test.hpp
|
```

Chaque figure de ce graphique correspond à un .cpp et un .hpp (même principe que un .c et un .h en c++). Cela soulève donc un premier point : beaucoup de fichier à gérer il va donc falloir créer tous les fichiers binaires (les .o) pour tout ces fichiers. Nous devons donc créer asserv.o, world.o, navigation.o, detection.o, protocole.o, affichage.o. Nous allons vouloir automatiser la création de ces binaires pour cela nous allons utiliser des commandes spéciales aux Makefile[1]. Pour automatiser la transformation des .cpp en .o nous avons les lignes suivantes

```
1 %.o: rasp/src/%.cpp
2     $(CC) $(CFLAGS) -c $<
```

Cela constitue la première règle de notre Makefile, elle permet de transformer n'importe quel .cpp dans code/rasp/src/ en un .o (qui sera mis dans code/). Nous voyons apparaitre deux variables inconnues "$(CC)" et "$(CFLAGS)", ces variables seront définies en début de Makefile et permettent de régler le problème 6. En effet "CC" va contenir le compilateur et "CFLAGS" les options de compilation. Nous ajustons donc notre Makefile comme suit.

```
1 CFLAGS = -Wall -Wextra -std=c++11 -g -O3
2 CC=g++
```

1. Je passe sous silence les commandes spéciales, si vous voulez des précision demandez à Lucas H. ou moi

```
3  %.o: rasp/src/%.cpp
4      $(CC) $(CFLAGS) -c $<
```

Maintenant que nous avons obtenu un moyen de transformer tous les fichiers .cpp en fichier binaire .o nous allons voir comment créer nos deux executables que nous appelerons project (qui contiendra le projet en entier) et le test (qui contiendra le test à effectuer) nous créons alors deux règles :

```
1  CFLAGS = -Wall -Wextra -std=c++11 -g -O3
2  FILE=navigation.o detection.o protocole.o asserv.o world.o affichage.o
3  CC=g++
4
5  all: project
6
7  %.o: rasp/src/%.cpp
8      $(CC) $(CFLAGS) -c $<
9
10 project: $(FILE) rasp/src/main.cpp
11     $(CC) $(CFLAGS) $(FILE) rasp/src/main.cpp -o project
12
13 test: $(FILE) rasp/tst/test.cpp
14     $(CC) $(CFLAGS) $(FILE) rasp/tst/test.cpp -o test
```

Nous voyons alors une nouvelle variable apparaitre, la variable “$FILE”, cette dernière va lister les dépendances (c’est à dire tous les fichier .o nécessaire pour la réalisation des règles). A ce stade il ne reste plus beaucoup de chose à faire sur le Makefile, si vous avez tout compris jusque là vous pourrez comprendre et créer un Makefile sans trop de soucis. Il faut encore rajouter deux règles qui sont très souvent présentes dans les Makefile “all” et “clean” (qui permet de nettoyer le dossier). Nous fournissons alors le Makefile final qui permet de réaliser le projet. Ce Makefile est totalement générique et est réutilisable pour tous vos projets.

```
1  CFLAGS = -Wall -Wextra -std=c++11 -g -O3
2  FILE=navigation.o detection.o protocole.o asserv.o world.o affichage.o
3  CC=g++
4
5  ##
6  # Project Title
7  #
8  # @file
9  # @version 0.1
10
11 all: project
12
13 %.o: rasp/src/%.cpp
14     $(CC) $(CFLAGS) -c $<
15
```

```
16 project: $(FILE) rasp/src/main.cpp
17     $(CC) $(CFLAGS) $(FILE) rasp/src/main.cpp -o project
18
19 test: $(FILE) rasp/tst/test.cpp
20     $(CC) $(CFLAGS) $(FILE) rasp/tst/test.cpp -o test
21
22 clean:
23     rm -f *.o project test test_protocole
24
25 # end
```