Projet Techno (4TIN403U)

Compilation

Moodle : https://moodle1.u-bordeaux.fr/course/view.php?id=10858

Responsable : Aurélien Esnard

Compilation : Hello World !

Exemple d'un petit programme C

```
// hello.c
#include <stdio.h>
int main(void) {
  printf("hello world!\n");
  return 0;
}
```

Compilation : le compilateur *gcc* produit un exécutable *a.out* à partir du fichier source

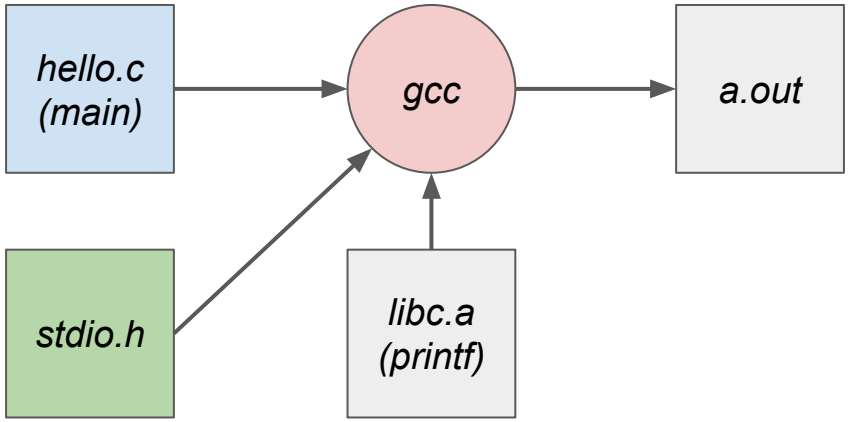
```
$ gcc hello.c
```

Exécution

```
$ ./a.out
hello world!
```

Compilation

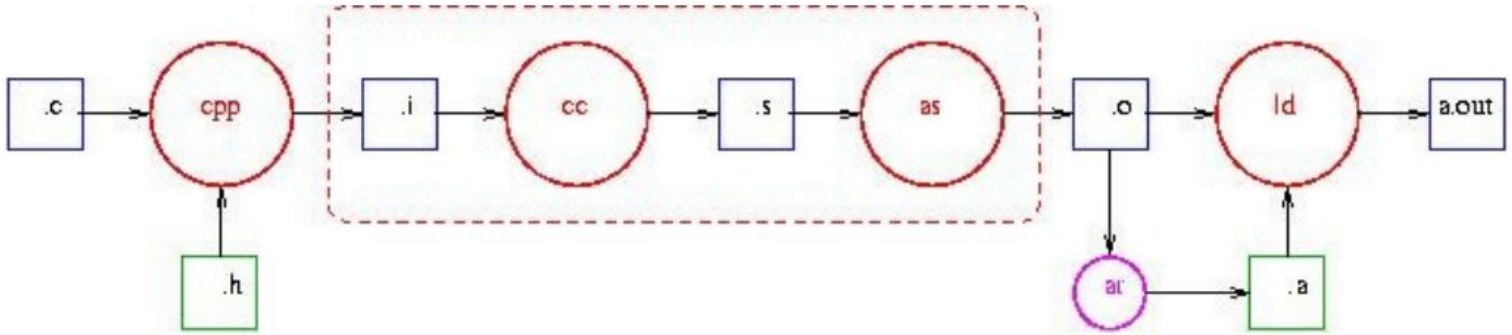
Ce n'est pas si simple…



- *hello.c* : fichier source avec la fonction *main*
- *stdio.h* : fichier entête (*header*) → déclaration de la fonction *printf*
- *libc.a* : bibliothèque C → incluant le code objet de la fonction *printf*
- *a.out* : fichier exécutable, produit par le compilateur *gcc*

Compilation

En fait, c'est même complexe !



Compilation de *gcc* en plusieurs phases

- phase de *prepocessing* (*cpp*, *gcc -E*, génère des .i)
- phase de compilation en langage assembleur (*cc, gcc -S*, génère des .s)
- phase assembleur (*as, gcc -c*, génère une fichier objet .o)
- phase d'édition de lien (*ld*, *gcc -o*, génère un exécutable)

Compilation

Compilation avec le standard C99 et le nom de l'exécutable

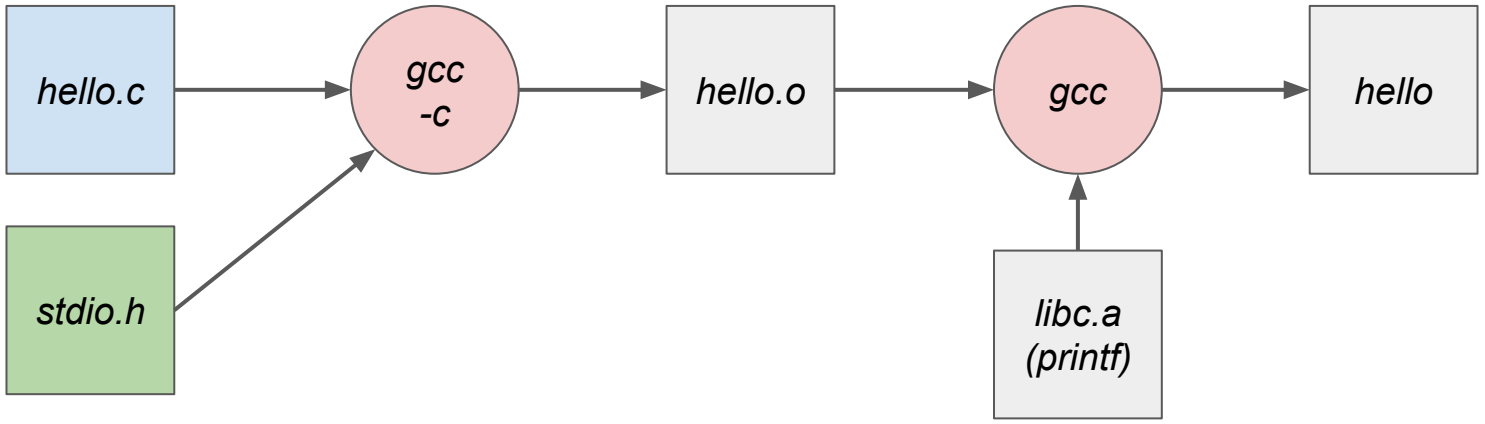
```
$ gcc -std=c99 hello.c -o hello
$ ./hello
```

Compilation

```
$ gcc -std=c99 -c hello.c      # compilation (génère hello.o)
```

Génération de l'exécutable

```
$ gcc hello.o -o hello         # édition de lien (génère hello)
```



Compilation : un autre exemple

Un programme monolithique avec deux fonctions *foo* et *bar*

```
// foobar.c
#include <stdio.h>

int foo(int x) {
    return ++x;
}

double bar(double x) {
    return x*x;
}

int main(void)
{
    int x = foo(4);
    double y = bar(x);
    printf("foo=%d, bar=%g\n", x, y);
    return 0;
}
```

Compilation Séparée

Découpage d'un programme "trop long" en plusieurs fichiers…

```
// foobar.c
#include <stdio.h>
int foo(int x);          // décalaration de foo()
double bar(double x);    // déclaration de bar()

int main(void)
{
    int a = foo(4);
    double b = bar(a);
    printf("foo=%d, bar=%g\n", a, b);
    return 0;
}
```

La déclaration d'une fonction est une *promesse* au compilateur de lui fournir son code dans d'autres fichiers (*foo.c* & bar.c)

Compilation

```
$ gcc -std=c99 -c foobar.c    # compilation (génère foobar.o)
```

Compilation Séparée

La suite du code, en deux fichiers séparés…

```
// foo.c
int foo(int x)
{
    return ++x;
}

// bar.c
double bar(double x)
{
    return x*x;
}
```

Compilation (suite)

```
$ gcc -std=c99 -c foo.c    # compilation (génère foo.o)
$ gcc -std=c99 -c bar.c    # compilation (génère bar.o)
```

Compilation Séparée

Résumé : compilation de fichiers séparés

```
$ gcc -std=c99 -c foo.c       # compilation (génère foo.o)
$ gcc -std=c99 -c bar.c       # compilation (génère bar.o)
$ gcc -std=c99 -c foobar.c    # compilation (génère foobar.o)
```

Ou plus simplement

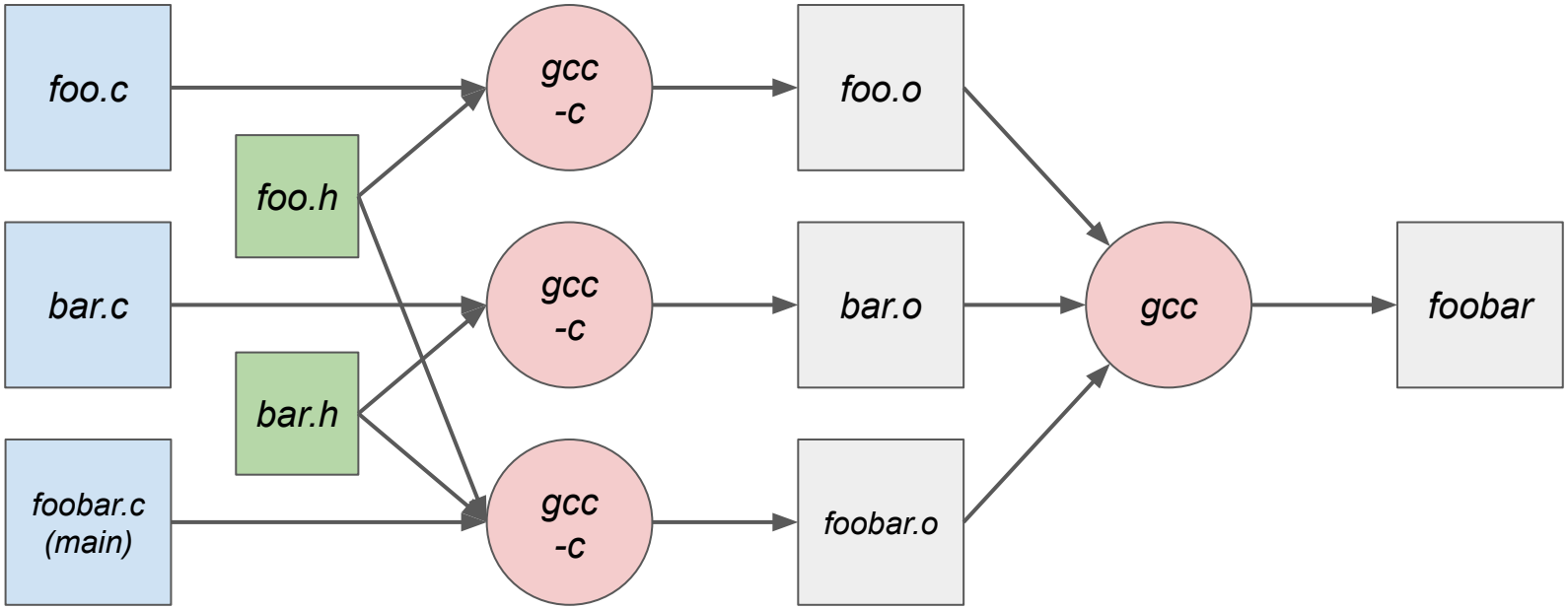
```
$ gcc -std=c99 -c foo.c bar.c foobar.c   # compilations (*.c)
```

Génération de l'exécutable

```
$ gcc foo.o bar.o foobar.o -o foobar     # édition de lien (*.o)
```

Compilation Séparée

En résumé…



Motivation – Un meilleur découpage de son code en plusieurs fichiers ; une réduction du temps de compilation en ne re-compilant que les fichiers sources modifiés et non tout le code !

Utilisation d'un fichier entête

Regrouper la déclaration de fonctions dans un fichier entête ou *header* (.h)

```
// foo.h
int foo(int x);

// bar.h
double bar(double x);
```

Inclure les .h dans les .c où il sont utiles avec la directive *#include*

```
// foobar.c
#include "foo.h"          // décalaration de foo()
#include "bar.h"          // déclaration de bar()

int main(void)
{
    int a = foo(4);
    double b = bar(a);
    printf("foo=%d, bar=%g\n", a, b);
    return 0;
}
```

Les Bibliothèques

Bibliothèque (ou *library*) : un fichier .a qui regroupe plusieurs fichiers objets .o

Utilisation d'un bibliothéque statique standard (libm.a → option -lm)

```
$ gcc foo.o bar.o foobar.o -o foobar -lm # bar.c utilse sqrt()
```

Génération d'une bibliothèque statique *fb* (libfb.a)

```
$ gcc -std=c99 -c foo.c bar.c      # génération de foo.o et bar.o
$ ar rcs libfb.a foo.o bar.o       # génération de libfb.a
```

Utilisation d'une bibliothèque statique "maison" (libfb.a → option -lfb)

```
$ gcc -std=c99 -c foobar.c         # génération de foobar.o (main)
$ gcc foobar.o libfb.a -o foobar   # génération de l'exécutable
```

Ou de manière plus élégante avec -lfb (et -L.)

```
$ gcc foobar.o -o foobar -lfb -L.  # génération de l'exécutable
```

Compilation

Memento sur les options du compilateur *gcc*

- -c : option pour compiler des fichiers source (.c) en fichier objet (.o)
- -std=c99 : spécifie le standard que le code source satisfait
- -o <fichier> : génère l'exécutable dans <fichier> au lieu de a.out
- -g : place dans l'exécutable des informations utiles au débuggeur
- -O2 : provoque certaines optimisations (de niveau 2) du code généré
- -Wall : (Warning ALL) affiche tous les message d'avertissement sur votre code
- -Werror : transformer les *warnings* en *errors*
- -I <rep> : ajoute <rep> à la liste des répertoires consultés pour trouver les fichiers entête (.h) inclus avec la directive #include "fichier.h"
- -l<nom> : inclut la bibliothèque *lib<nom>.a* lors de la génération de l'exécutable (la bibliothèque *libc.a* est implicitement inclut)
- -L <rep> : ajoute <rep> à la liste des répertoires consultés pour trouver les fichiers .a des bibliothèques
- Pour toutes les autres options, lire le manuel : *man gcc*

Makefile

Makefile

Principe : Le fichier *Makefile* (associé à l'outil *make*) est une solution élégante pour compiler automatiquement et efficament un projet de programmation.

On écrit dans un fichier *Makefile* des règles de la forme suivante :

```
cible1 : dep1 dep2 dep3 …
    commande …

dep1 : autre_dep1 autre_dep2 …
    autre_commande …
```

Attention, à la tabulation obligatoire avant chaque commande !

Compilation : La compilation du projet est déclenché par l'appel à la commande *make* dans le même répertoire…

```
$ make          # compilation de la première cible (ALL, en général)
$ make cible1   # compilation d'une cible particulière
```

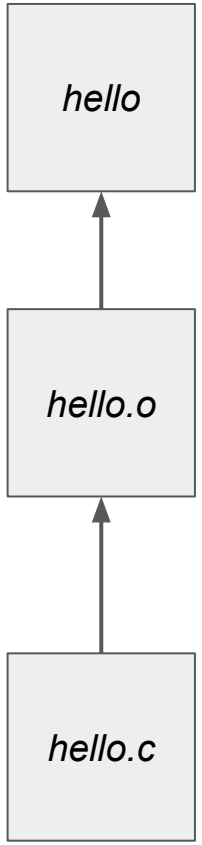
Makefile : Hello World !

Considérons un code minimaliste : *hello.c*

```
#include <stdio.h>
int main(void) {
  printf("hello world!\n");
  return 0;
}
```

Voici un Makefile simple et explicite…

```
ALL : hello

hello : hello.o
    gcc hello.o -o hello

hello.o : hello.c
    gcc -c hello.c

clean:
    rm -f hello.o hello
```



Makefile : Hello World !

Et voici un second Makefile minimaliste utilisant les règles implicites…

```
ALL : hello
hello : hello.o
hello.o : hello.c
```

Compilation

```
$ make
cc    -c -o hello.o hello.c
cc   hello.o   -o hello
```

Règles génériques et implicites (*builtin*) de la forme suivante…

```
# génération d'un fichier .o à partir d'un fichier.c du même nom
%.o : %.c
        $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

Makefile : Hello World !

Utilisation des variables *builtin* : CC, CFLAGS, CPPFLAGS, …

```
CC= gcc
CFLAGS= -std=c99 -Wall -O2

ALL : hello
hello : hello.o
hello.o : hello.c
```

Compilation

```
$ make
gcc -std=c99 -Wall -O2   -c -o hello.o hello.c
gcc   hello.o   -o hello
```

Makefile : Compléments

Commentaires

Les commentaires sont précédés du symbole #.

Variables contextuelles ecrire des règles génériques

- La variable $@ représente la cible courante
- La variable $^ représente la liste des dépendances
- La variable $< représente la première dépendance

Divers

La cible .PHONY permet d'indiquer des cibles particulières qui ne sont pas des fichiers, comme par exemple la cible *clean*.

Makefile : Pour aller plus loin

Makefile récursif

```
make -C subdir
```

Wildcard

```
SOURCES  := $(wildcard *.c)
INCLUDES := $(wildcard *.h)
OBJECTS  := $(SOURCES:.c=.o)
```

Quelques astuces

- @ suppresses the normal 'echo' of the command that is executed.
- - means ignore the exit status of the command that is executed (normally, a non-zero exit status would stop that part of the build).

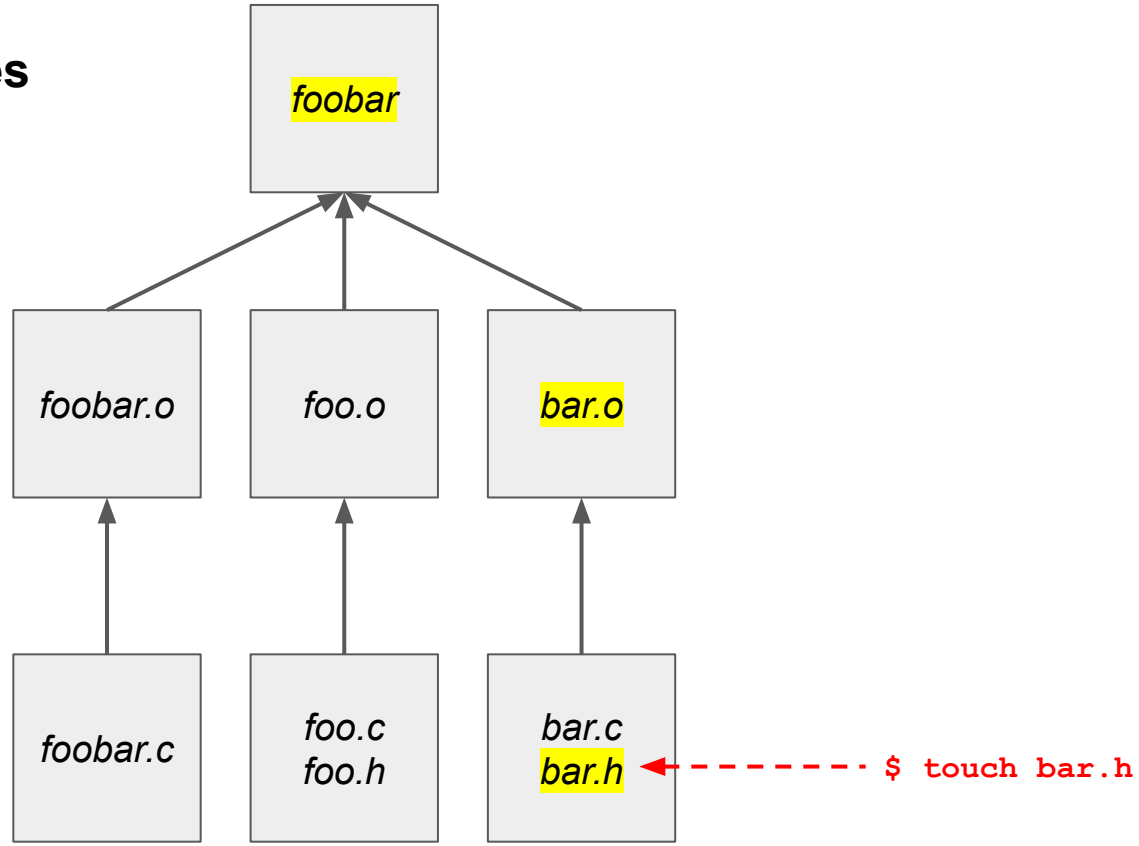
```
clean :
    @echo "clean all"
    -rm *.o                    # sinon utiliser rm -f (force)
```

Makefile : Exemple foobar

```
CC= gcc
AR= ar
CPPFLAGS=
CFLAGS= -Wall -std=c99
LDFLAGS= -L.
LDLIBS= -lfb -lm

ALL : foobar libfb.a

libfb.a : bar.o foo.o
    $(AR) rcs $@ $^

foobar : foobar.o libfb.a
    $(CC) $(LDFLAGS) foobar.o -o $@ $(LDLIBS)

foobar.o : foobar.c bar.h foo.h
bar.o : bar.c bar.h
foo.o : foo.c foo.h

.PHONY : clean
clean :
    rm -f *.o *.a foobar
```

Makefile : les dépendances

L'arbre des dépendances



Nota Bene : Si un fichier source est modifié (par exemple avec *touch*), la commande *make* analyse les dépendances pour ne recompiler que ce qui est utile (en comparant la date de dernière modification des fichiers sources vs les cibles).

Makefile : les dépendances

Comment découvrir automatiquement les dépendances de compilation ?

```
$ gcc -MM *.c
bar.o: bar.c
foobar.o: foobar.c foo.h bar.h
foo.o: foo.c
```

Et simplifions un peu notre Makefile foobar en exprimant juste les dépendances (sans la commande qui est implicite)

```
# …

foobar.o: foobar.c foo.h bar.h
foo.o: foo.c foo.h
bar.o : bar.c bar.h

# …
```

Makefile : pour aller plus loin !

Tutoriel

⇒ https://aurelien-esnard.emi.u-bordeaux.fr/teaching/doku.php?id=projtec:make

Documentation

⇒ https://www.gnu.org/software/make/manual/