Compilation

Plan

- **gcc (compilation et compilation séparée)**
- **Make (Makefile)**
- **Makefile générique**

Compilation simple

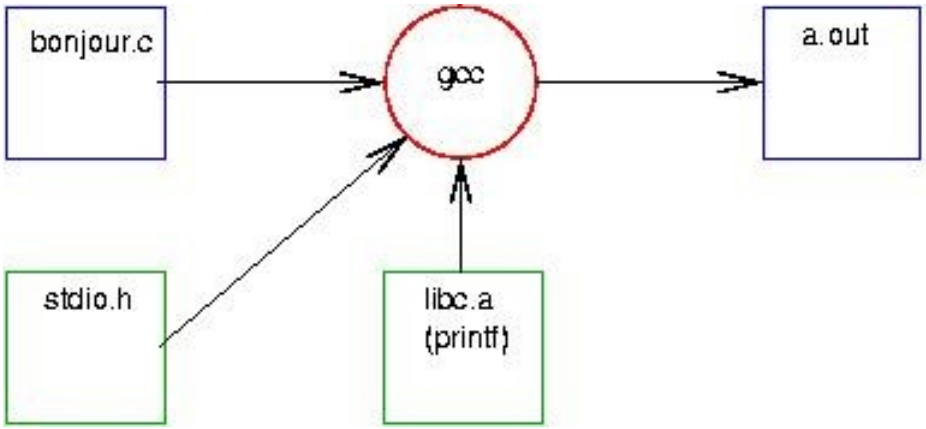
```
#include <stdio.h>

#define VALEUR 5

int main(void) {
  int i = VALEUR;
  printf("Bonjour a tous %d\n", i) ;
  return 0 ;
}
```

La commande

gcc bonjour.c

Produit un fichier a.out. Qui peut être exécuté.

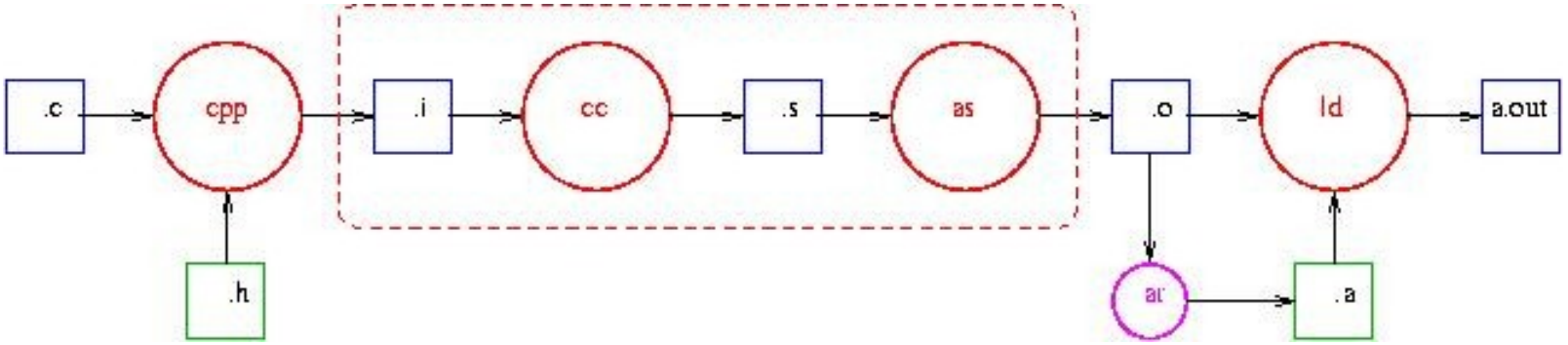
Compilation simple (pas si simple)



Pour générer le fichier `a.out`, `gcc a` également besoin :

- du fichier `stdio.h`
- du fichier `libc.a` (pour la fonction `printf`)

Compilation simple (pas si simple)



La « compilation » se déroule en fait en 3 phases :

1. Le préprocesseur (cpp)
2. La compilation à proprement parlé cc+ as
3. L'édition de lien (ld)

ar est une application extérieure à gcc qui permet d'archiver plusieurs objets (fichiers suffixés .o) en une bibliothèque (fichier suffixé .a).

gcc : Options

- Pour arrêter le compilateur a une certaine phase
  - -E : s'arrête à la premiere phase (génère des .i)
  - -S : s'arrête avant l'assembleur (génère des .s)
  - -c : s'arrête à la fin de la second phase (génère des .o)

Pour reprendre le travail depuis une certaine phase gcc accepte plusieurs fichiers d'entrée sur la ligne de commande et, pour chacun d'eux, en fonction du suffixe, leur applique uniquement les phases nécessaires. La dernière phase (l'édition de liens) n'est exécutée qu'une fois, globalement, près avoir effectué toutes les phases précédentes nécessaires sur tous les fichiers.

gcc : Autres options

- Pour augmenter le contexte de la compilation
  - -I<rep> : ajoute <rep> à la liste des répertoires consultés par le préprocesseur pour trouver les fichiers cités dans les commandes de la forme `#include <...>`
  - -l<nom> : invite l'éditeur de liens à considérer la bibliothèque lib<nom>.a pour y rechercher la définition d'identificateur utilisé et pas défini.
    - `libc.a` est toujours considérée
  - -L<rep> : ajoute <rep> à la liste des répertoires consultés par l'éditeur de liens pour trouver les bibliothèques à considérer.

Options diverses

- -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 débugger (cf. chapitres suivants)
- -O2 : provoque certaines optimisations du code généré
  - Rq : -g ou -02 il faut choisir…
- -Wall : (Warning ALL) affiche un message pour toute forme un peu ambiguë relevée dans le code.
- …

Compilation séparée

- C’est quoi ?
  - Découper le code en plusieurs fichiers
  - Compiler chaque fichier séparément
- Pourquoi découper le code en plusieurs fichiers ?
  - Plus lisible
  - Plus modulaire
  - Plus facile de travailler à plusieurs sur des fichiers différents
- Pourquoi re-compiler chaque fichier indépendamment ?
  - Plus rapide !!! (ex : compilation de libre office : plusieurs heures)
- => Outils pour générer la compilation séparée : make

Compilation séparée : exemple simple

- Pour développer un éditeur (`edit`)de texte en C on découpe le programme en plusieurs fichiers sources : kbd.c, display.c, files.c, command.c, main.c. Par ailleurs les définitions générales sont réparties sur 2 fichiers : defs.h et command.h.
- On trouve dans les fichiers : kbd.c, files.c, command.c les lignes suivantes :
  - `#include «defs.h »`
  - `#include « command.h »`
- On trouve dans les fichiers : display.c, main.c les lignes suivantes :
  - `#include «defs.h »`

Quelles commandes pour générer l'exécutable `edit` ? Quel graphe de dépendances ?

Makefile correspondant

```
edit : main.o kbd.o command.o display.o files.o
        gcc -o edit main.o kbd.o command.o \
                display.o files.o

main.o : main.c defs.h
        gcc -c main.c
kbd.o : kbd.c defs.h command.h
        gcc -c kbd.c
command.o : command.c defs.h command.h
        gcc -c command.c
display.o : display.c defs.h
        gcc -c display.c
files.o : files.c defs.h command.h
        gcc -c files.c

clean :
        rm edit main.o kbd.o command.o \
                display.o files.o
```

Syntaxe d'un fichier Makefile

- Des règles + des définitions de variables [+ des directives]
- Une règle est constituée
  - d'un ensemble de cibles,
  - d'un ensemble de dépendances et
  - d'une séquence de commandes.
  - Syntaxe :

```
cible cible ... : dependance dependance ...
	commande
	commande
```

Tabulation

  - make interprète ces règles de la façon suivante. Pour chaque fichier cible, s'il n'existe pas, ou si sa date de dernière modification est plus ancienne que la date de dernière modification d'un des fichiers dépendances, alors il doit être construit, ou reconstruit, en exécutant successivement les commandes.

Syntaxe d’un fichier Makefile

- Définition de variable :
  - Syntaxe :

```
<VARIABLE> = <EXPRESSION>
```

  - Exemple 1 :

```
OBJETS = main.o kbd.o command.o display.o files.o
edit : $(OBJETS)
        gcc -o edit $(OBJETS)
```

  - Exemple 2 :

```
#Liste des fichiers *.c
SOURCES = $(wildcard *.c)

#remplace .c en .o dans la liste
OBJETS = $(SOURCES:.c=.o)
```

Syntaxe d'un fichier Makefile

- Variables implicites :
  - CC : C Compiler (défaut : cc)
  - CXX : C++ Compiler (défaut : g++)
  - CFLAGS (options de compilation)
  - CPPFLAGS (options du préprocesseur)
  - LDFLAGS (options de l'éditeur de lien)
  - ...
- Exemples :

```
CC = gcc
CFLAGS = -g –Wall –std=c99
CPPFLAGS = -I ../include
LDFLAGS = -lm
```

Syntaxe d’un fichier Makefile

- Directives :
  - Instruction à l’intention du « préprocesseur » de make
  - Exemple 1 :

```
include ../Makefile.vars
```

  - Exemple 2 :

```
ifeq ($(DEBUG),YES)
  CFLAGS = -g –Wall –std=c99
else
  CFLAGS = -02 –Wall –std=c99
endif
```

Syntaxe d’un fichier Makefile

- Variables automatiques (valables dans les commandes) :
  - $@ The file name of the target of the rule.
  - $< The name of the first dependency.
  - $? The names of all the dependencies that are newer than the target, with spaces between them.
  - $^ The names of all the dependencies, with spaces between them.
- Exemple d’utilisation

```
kbd.o : kbd.c defs.h command.h
    $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

Makefile V2

```
CFLAGS = -g –Wall –std=c99
CPPFLAGS = -I ../include
SOURCES = $(wildcard *.c)
OBJETS = $(SOURCES:.c=.o)
edit : $(OBJETS)
        $(CC) -o $@ $^

main.o : main.c defs.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
kbd.o : kbd.c defs.h command.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
command.o : command.c defs.h command.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
display.o : display.c defs.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
files.o : files.c defs.h command.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
clean :
        rm edit $(OBJETS)
```

Makefile V2

```
main.o : main.c defs.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
kbd.o : kbd.c defs.h command.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
command.o : command.c defs.h command.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
display.o : display.c defs.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
files.o : files.c defs.h command.h
        $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

Rq: il existe des règles implicites pour make : «
-Compiling C programs *n*.o is made automatically from *n*.c
with a recipe of the form '$(CC) $(CPPFLAGS) $(CFLAGS) -c'.
-Compiling C++ programs *n*.o is made automatically from *n*.cc, *n*.cpp, or *n*.C
with a recipe of the form '$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c'.
-… »
Si une règle n'a pas de commande, make va chercher la règle implicite qui s'applique au motif.

Makefile V3

```
CFLAGS = -g –Wall –std=c99
CPPFLAGS = -I ../include
SOURCES = $(wildcard *.c)
OBJETS = $(SOURCES:.c=.o)

edit : $(OBJETS)
        $(CC) -o $@ $^

main.o : main.c defs.h
kbd.o : kbd.c defs.h command.h
command.o : command.c defs.h command.h
display.o : display.c defs.h
files.o : files.c defs.h command.h

.PHONY : clean #indique à make d'ignorer l'éventuel fichier clean.
clean :
        rm edit $(OBJETS)
```

Makefile générique

- Makefile V3 : doit être potentiellement modifié à chaque fois qu'on modifie des `#include`.
  - Sinon risque d'incohérence dans le processus de compilation…
  - « Je comprends pas, le programme ne correspond pas aux sources :-(»

- L'option « -M » de gcc permet de générer les règles de dépendances :

```
gcc –M $(CPPFLAGS) bonjour.c > depends.txt
```

Makefile V4

```
CFLAGS = -g –Wall –std=c99
CPPFLAGS = -I ../include
SOURCES = $(wildcard *.c)
OBJETS = $(SOURCES:.c=.o)

edit : $(OBJETS)
        $(CC) -o $@ $^

include depends.txt


.PHONY : clean
clean :
        rm edit $(OBJETS)
depends.txt : $(SOURCES)
   $(CC) –M $(CPPFLAGS) $(SOURCES) > depends.txt
```

What else

- `all` : cible par défaut (`make <=> make all`)
- `VPATH` : indique où chercher des fichiers.
  - Utile si les .h se trouvent dans un autre répertoire (par exemple ../include)
- `make -C <repertoire>` : se place dans <repertoire> avant de lancer make. -> Makefile recursif
- Peut bien évidemment être utilisé pour générer autre chose qu'un exécutable
  - Ex: document pdf à partir de sources LaTeX...
  - Intégrer la génération des tests et de la documentation
  - Publier des pages web
  - ...

Références

- Documentation sur gcc : http://gcc.gnu.org/onlinedocs/
- Documentation sur « Make » : http://www.gnu.org/software/make/manual/
  - Quick Références (http://www.gnu.org/software/make/manual/html_node/Quick-Reference.html#Quick-Reference)
  - Wildcards (http://www.gnu.org/software/make/manual/html_node/Wildcards.html#Wildcards)
  - Variables automatiques (http://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html#Automatic-Variables)
  - How to use variables (http://www.gnu.org/software/make/manual/html_node/Using-Variables.html#Using-Variables)
  - Implicite rules (http://www.gnu.org/software/make/manual/html_node/Implicit-Rules.html#Implicit-Rules)
  - gcc –M (http://www.gnu.org/software/make/manual/html_node/Automatic-Prerequisites.html)