Outils de la chaîne de compilation

- Commandes de base du shell
- Les outils de la chaîne de compilation utilisés
- Les arguments utilisés du compilateur
- Le Makefile forme générale et règles génériques
- Les 4 types de directives du préprocesseur du C

Commandes de base du shell

En TP, vous devez utiliser les commandes du shell pour:

- éditer les fichiers source et les fichiers de trace
- lancer le compilateur au travers d'un Makefile
- vous balader dans les répertoires,
- etc.

Normalement, cela fait partie de votre bagage de connaissances, mais si vous avez des lacunes, regarder

- le tutoriel d'open-classroom : https://link.infini.fr/shell
- une doc : https://lea-linux.org/documentations/Admin-admin_env-shell ou https://fr.wikibooks.org/wiki/Programmation_Bash/Commandes_shell

Chaîne de compilation GNU

GNU propose une chaîne d'outils de compilation permettant de produire un exécutable à partir de programmes source :

`gcc -c`	préprocesseur + compilateur (`gcc -c file.c` → `.o`)
`ld`	Éditeur de liens (ou `gcc file.o` → `.x`)

mais pas seulement, il y a d'autres outils :

`cpp`	préprocesseur seul (ou `gcc -E file.c` → `.c`, idem pour `.S`)
`as`	Assembleur seul (ou `gcc -S file.c` → `.s`)
`objdump`	désassembleur (elf `.o` ou `.x` → `.s`)
`gdb`	debugger (pour exécuter en pas à pas à condition que le code ait été compilé avec l'option -g, cela nécessite l'intervention du système d'exploitation)
`nm`	liste des symboles présents dans un fichier au format elf
`readelf`	affichage du contenu d'un fichier au format elf

Compilateur C : arguments utilisés

L'application gcc permet d'appeler le préprocesseur, le compilateur, l'assembleur et l'éditeur de liens. https://gcc.gnu.org/onlinedocs/gcc-7.1.0/gcc/ (gcc utilisé en TP) fait plus 900 pages. Nous n'allons voir que quelques arguments… (ceux utilisés en TME)

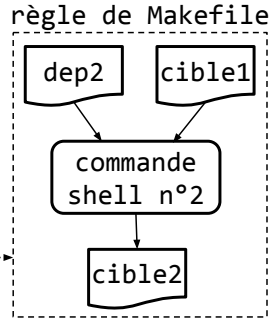
`gcc -c`	: stoppe gcc après le compilateur produit un .o ou un .s
`-Wall`	: tous les warnings du langage C (ou presque) sont relevés
`-Werror`	: tous les warnings sont considérés comme des erreurs
`-mips32r2`	: informe le compilateur du type de MIPS32 révision 2
`-std=c99`	: permet, entre autres, la déclaration des var. locales n'importe où
`-fno-common`	: ne pas utiliser une section common pour les var. glob. non-static
`-fno-builtin`	: ne pas utiliser les fonctions "builtin" de gcc (p. ex. `strcpy()`)
`-fomit-frame-pointer`	: demande d'utiliser $29 comme seul pointeur de pile
`-G0`	: ne pas utiliser de global pointer ($28) pour les variables globales
`-O3`	: demande l'optimisation maximale (il existe : `-O0, -O1, -O2, -Os`)
`-o <file>`	: nom du fichier de sortie `<file>`
`-I <dir1:dir2..>`	: ajoute `<dir1:dir2..>` aux répertoires de recherche des `#include`
`-T <ldscript>`	: informe l'éditeur de lien du nom du ldscript (si ld est invoqué)

Fichier `Makefile` & Commande `make`

Un `Makefile` est un fichier contenant la méthode de construction d'un fichier cible à partir de ces sources. Un fichier `Makefile` est interprété par la commande `make`

La commande `make` a deux objectifs *(l'objectif 1 est LE plus important)*

1. Décrire la méthode de construction permettant de la rejouer complètement ou en partie après un changement des sources.
2. Permettre une reconstruction sélective en n'exécutant que les étapes de construction nécessaires pour produire la cible lorsque seule une partie des sources a été modifiée



Un fichier `Makefile` est constitué d'un ensemble de règles

`Makefile`

```
cible1 : dep1
← TAB →commandes shell n°1
cible2 : cible1 dep2
← TAB →commandes shell n°2
```

puisque `cible2` dépend de `cible1` alors

```
bash
$> make cible2
   commandes shell n°1
   commandes shell n°2
```

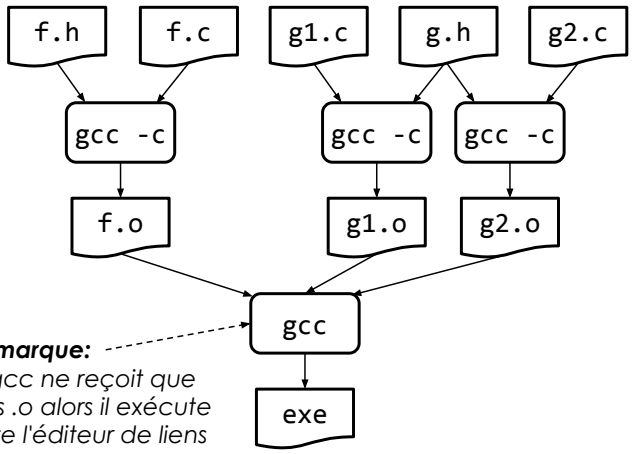
Vous verrez pendant les TME 3 manières d'utiliser un Makefile :
(1.) collection de scripts ; (2.) avec des règles explicites ; (3.) avec des règles implicites

1. Makefile : collection de *shell scripts*

Le Makefile peut contenir l'ensemble des *shell scripts* nécessaires à produire l'exécutable.
Le Makefile contient alors autant de règles qu'il y a scripts à réaliser :

- 1 pour produire l'exécutable,
- 1 pour faire le ménage,
- 1 pour exécuter, etc.

L'appel de `make` sans cible prend par défaut la première cible



Remarque:
Si gcc ne reçoit que des .o alors il exécute juste l'éditeur de liens

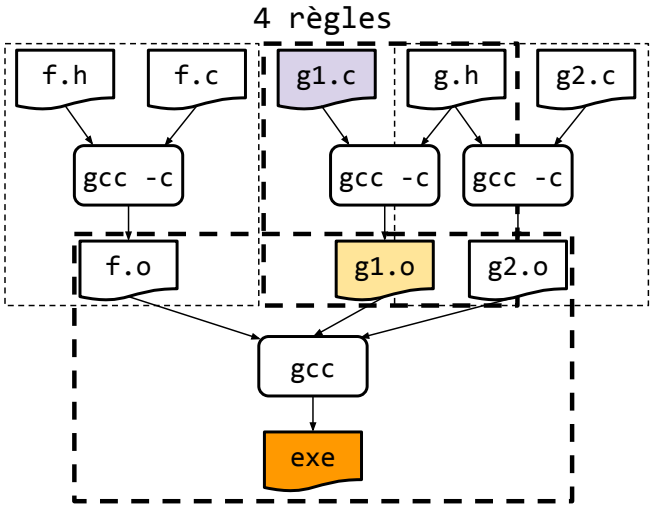
```
help: # regle par defaut qui affiche une aide
      @echo "compil    : compilation exec"
      @echo "clean     : efface les .o"
      @echo "cleanall  : tout sauf les sources"
      @echo "exec      : test du programme"

compil : # regle pour generer l'executable
      gcc -c -Wall f.c
      gcc -c -Wall g1.c
      gcc -c -Wall g2.c
      gcc f.o g1.o g2.o -o exe

exec: # regle pour lancer et tester l'executable
      exe arg1 arg2

clean:
      rm *.o

cleanall: clean
      rm exe
```

```
bash
$> make compil
   gcc -c Wall f.c
   gcc -c ...
```

2. Makefile : règles explicites

make lit d'abord le Makefile entièrement,
puis construit le graphe de construction,
puis produit la cible demandée uniquement
si sa date en antérieure à l'une de ces dépendances.

*Ici la règle par défaut est **dep** qui n'est pas un fichier*



Makefile

```
dep: exe

all : clean dep

f.o : f.c f.h
      gcc -c -Wall f.c

g1.o : g1.c g.h
      gcc -c -Wall g1.c

g2.o : g2.c g.h
      gcc -c -Wall g1.c

exe : f.o g1.o g2.o
      gcc f.o g1.o g2.o -o exec

clean :
      rm *.o
```

si seul **g1.h** a été modifié →

```
bash
$> make exe
   gcc -c -Wall g1.c
   gcc f.o g1.o g2.o -o exec
```


Makefile : règles génériques et variables

Le format Makefile est très riche (https://www.gnu.org/software/make/manual/)
(tuto:https://codes-sources.commentcamarche.net/faq/87-shell-linux-creation-des-makefiles-commande-make)

- Il permet d'exprimer des règles génériques
  - La dépendance de la cible utilise le caractère %
    qui représente un nom de fichier quelconque
  - Les commandes utilisent des variables automatiques
    dont la valeur est extraite de la ligne de dépendance
    - $@ : cible
    - $< : première dépendance
    - $^ : toutes les dépendances
    - $* : %
- on peut ajouter des règles de commandes
  - clean ou all et le .PHONY permet de dire à
    make que ces cibles ne sont pas des fichier
- on peut utiliser des variables
  - CFLAGS, BIN, etc.

```
CFLAGS = -Wall
OBJ    = f.o g1.o g2.o
BIN    = exe
.PHONY = all clean

all    : clean $(EXE)

%.o    : %.c
     gcc -c $(CFLAGS) $<

$(BIN) : $(OBJ)
     gcc $^ -o $@

clean  :
     rm $(OBJ)

f.o    : f.c f.h
g1.o   : g1.c g.h
g2.o   : g2.c g.h
```

1. Préprocesseur : expansion de macro

(https://gcc.gnu.org/onlinedocs/cpp/Macros.html)

```
#define   MACRO
#define   MACRO                 DÉFINITION
#define   MACRO(a1,a2,..,an)    DÉFINITION_AVEC_ARGUMENTS
#undef    MACRO
```

- Attention, on ne peut pas mettre de commentaire derrière une définition
- Une définition de macro est sur une seule ligne ou utiliser le caractère \
- Exemples

```
#define DEBUG
#define ROUGE 4
#define MAX(a,b)   ((a)>(b)?(a):(b))          → noter les ( )
#define INCV(v)    do{v.x++;v.y++;}while(0)   → si plusieurs instructions dans
[…]                                             la définition de la macro
struct v_st {int x, y} v1, v2;
if (MAX(4, 2*i+j) < ROUGE)
     INCV(v1);
else
     INCV(v2);
[…]
```


2. Préprocesseur : inclusion de fichiers

(https://gcc.gnu.org/onlinedocs/cpp/Header-Files.html)

`#include` permet d'inclure un fichier dans un autre

- `#include <fichier.h>` ou `#include "fichier.h"`
  - inclut *fichier.h* dans le fichier contenant la directive `#include`
  - avec " " : *fichier.h* est recherché dans le répertoire courant
  - avec < > : *fichier.h* est recherché dans les répertoires standards tel que /usr/include et dans les répertoires donnés par l'argument `-I`
- Bon usage :
  - N'inclure que des .h ⇒ **jamais** des .c
  - Se prémunir contre la double inclusion (évite la redéfinition des macros)

```
#ifndef _FICHIER_H_
#define _FICHIER_H_
    …
#endif
```

3. Préprocesseur : compilation conditionnelle

(https://gcc.gnu.org/onlinedocs/cpp/Conditionals.html)

Permet de sélectionner le code à inclure par exemple pour adapter le code à la machine ou traiter le debug

`Directives : #if  #ifdef  #ifndef  #else  #elif  #endif`

```
#ifdef DEBUG
    code 1
#else
    code 2
#endif
```

```
#if (defined DEBUG)
    code 1
#else
    code 2
#endif
```

```
#if VERBOSE > 1
    code 1
#elif VERBOSE > 0
    code 2
#else
    code 3
#endif
```

```
#if 0
    code
#endif
```

```
#if !(defined __MIPS__ || defined MIPS32)
    code 1
#endif
```


4. Préprocesseur : autres directives et macros

Contrôle de la compilation : (https://gcc.gnu.org/onlinedocs/cpp/Diagnostics.html)

```
#error   "message"    : affiche "message" et stoppe la compilation
#warning"message"     : affiche "message"
```

Macros prédéfinies :
(https://gcc.gnu.org/onlinedocs/cpp/Standard-Predefined-Macros.html)

```
__FILE__              : MACRO contenant le nom du fichier courant
__LINE__              :   "       "     le numéro de ligne
__DATE__              :   "       "     la date
__TIME__              :   "       "     l'heure
__FUNC__              :   "       "     la fonction courante
```

Mise entre guillemets : (https://gcc.gnu.org/onlinedocs/cpp/Stringizing.html)

```
#define str(s)      #s      : str(bonjour)   → "bonjour"
```

Concaténation de 2 arguments :
(https://gcc.gnu.org/onlinedocs/cpp/Concatenation.html)

```
#define concat(a,b)  a##b     : concat(A,B)      → AB
```