TELECOM ParisTech

Institut Mines-Télécom

Introduction Langage C et Systèmes d’exploitation Partie 3: Langage C – Chaine de production

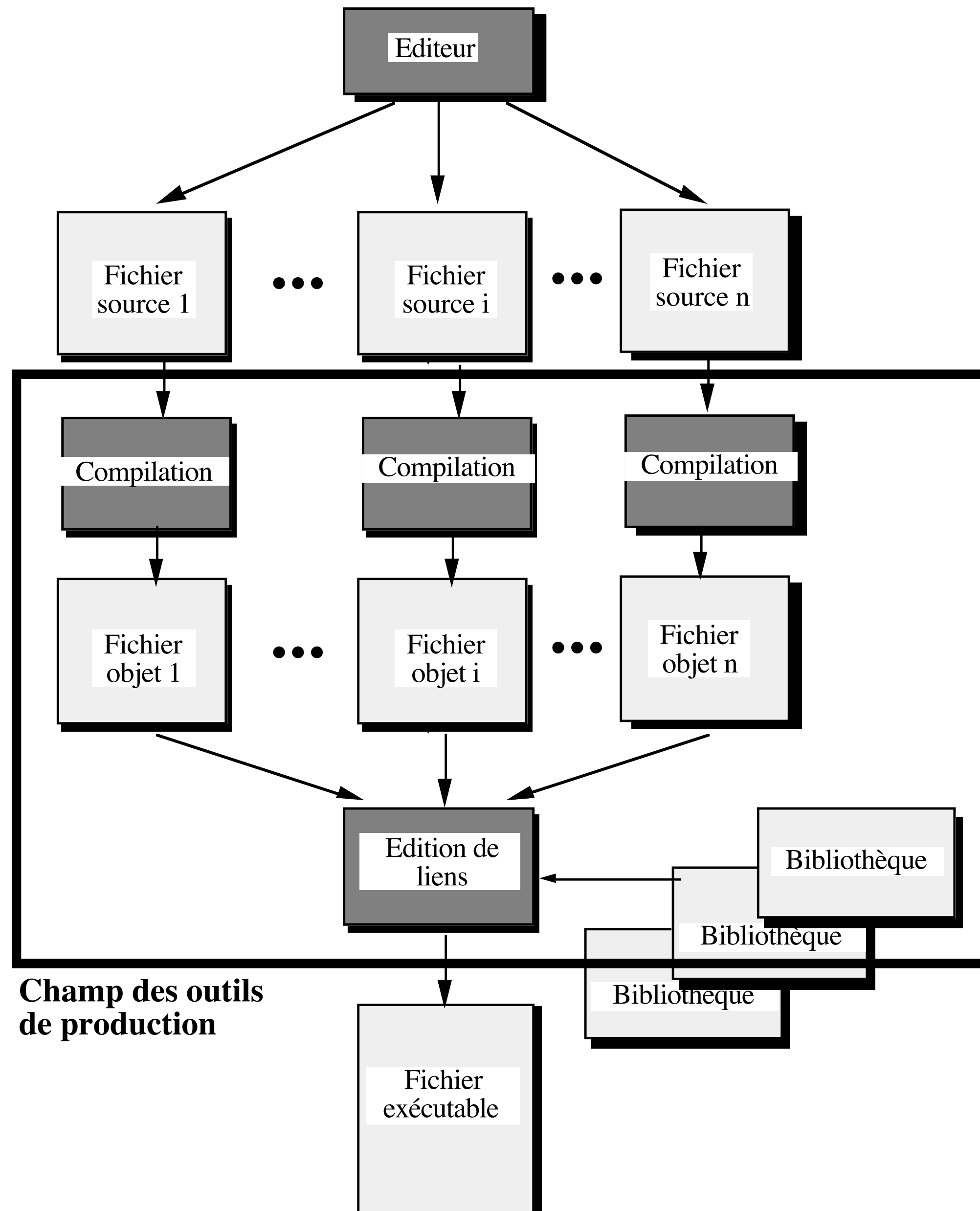
Responsable : Etienne Borde (C218-3)

Auteurs : Bertrand Dupouy et Etienne Borde

1. *Généralités :*
   - *Rappels: compilation séparée*
   - *Objectifs*

2. l'outil make
   - *cible, dépendance*
   - *fichier Makefile de base*
   - *Makefiles et variables*
   - *Makefile et règles implicites*
   - *Structuration des Makefile*
   - *un exemple plus sophistiqué pour la gestion des dépendances au fichiers .h*

Compilation séparée, rappels :



TELECOM ParisTech

Compilation séparée: problèmes potentiels

Si on utilise deux fichiers:

1. `f1.c` qui contient la fonction `main`
2. `f2.c` qui contient les fonctions utilisées par `main` :

On peut créer l'exécutable appli comme suit:

```
gcc -c f1.c
gcc -c f2.c
```

Compilations séparées des fichiers source et productions de deux fichiers objets.

```
gcc f1.o f2.o -o appli
```

Edition de liens à partir des deux fichiers objets et production d'un fichier exécutable appelé `appli`.

Problème: Bob modifie f1.c et f2.c
→ que se passe-t-il s'il oublie de recompiler f2.c?

Un exécutable est créé, mais il est probablement défectueux!

TELECOM ParisTech

Comment faciliter la production d'un exécutable?

- **Principe: automatiser les étapes (compilation/édition de liens) de façon intelligente**
  - « automatiser » = enchainer les étapes
  - « de façon intelligente » = (i) ne faire **que** ce qui est nécessaire et (ii) faire **tout** ce qui est nécessaire. Autrement dit, ne rien oublier mais ne rien faire d'inutile…

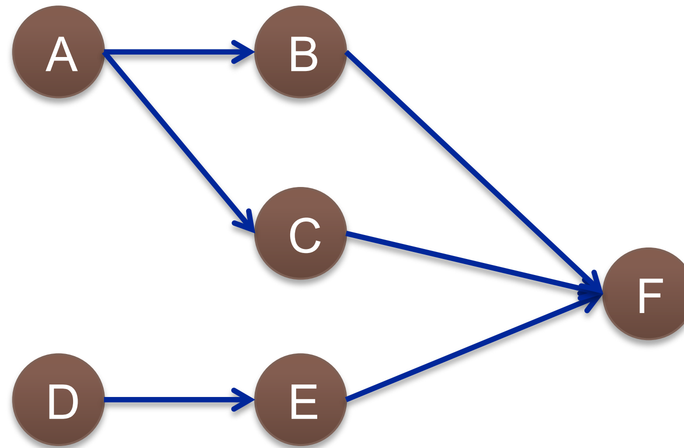
- **Automatiser: programmer la production du logiciel**
  - En C? … En Java? … Non, rassurez vous ;)
  - En utilisant des outils dédiés, ici l'outil « make »

- **De façon intelligente: via un graphe de dépendances**
  - Graphe orienté acyclique
  - Ne traiter un sommet du graphe que si les entrées dont ce sommet dépend ont changé

TELECOM ParisTech

Graphe de dépendances

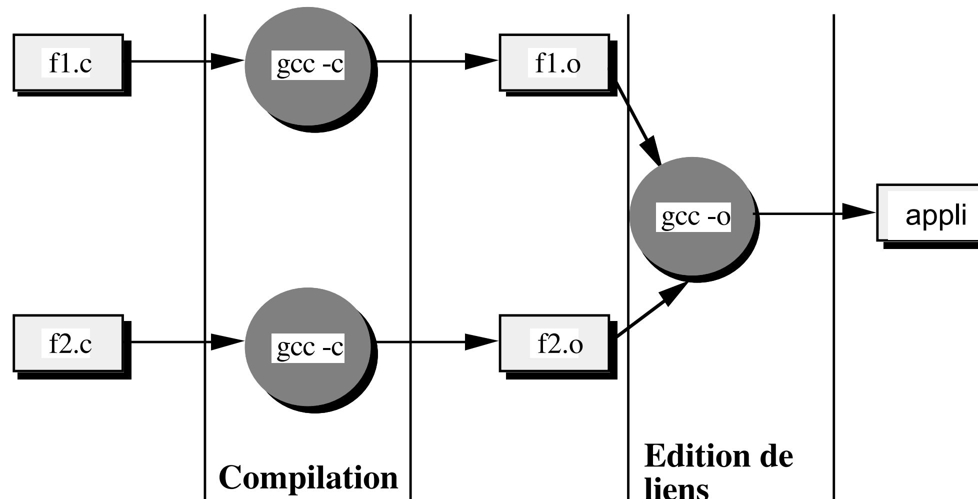
- **Sommet A = tâche à effectuer**
- **Arc (A,B) = B doit être effectuée si A a été effectuée**



- **Lorsque A est effectuée, B C et F doivent l'être…**
  - On identifie donc ce qui doit être fait et seulement ce qui doit être fait

Compilation séparée et graphe de dépendances

Production d'un fichier exécutable appli à partir des fichiers sources f1.c et f2.c :

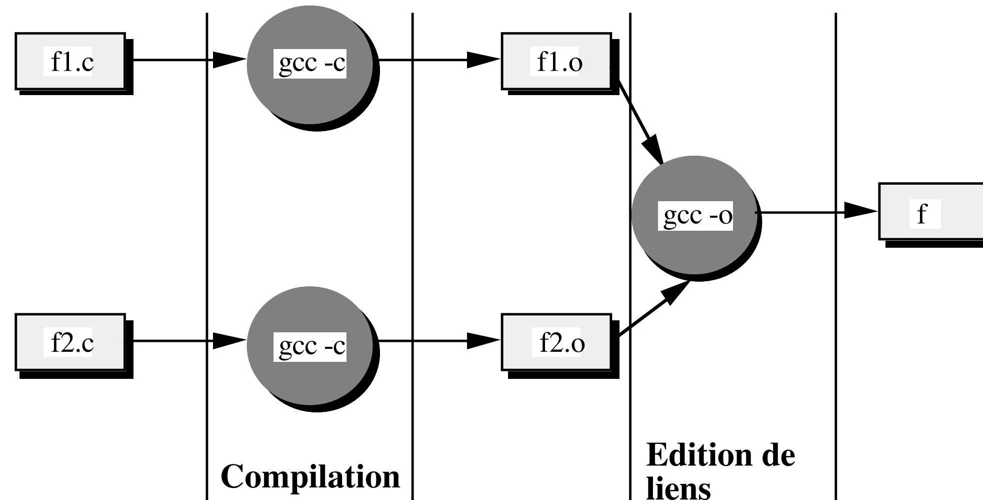


- Utilisation de gcc dans le schéma ci-dessus :

`gcc -c f1.c` (produit l'objet `f1.o`)
`gcc -c f2.c` (produit l'objet `f2.o`)
`gcc f1.o f2.o -o appli` (l'exécutable s'appelle `appli` au lieu `de a.out`)

TELECOM ParisTech

Comment savoir ce qu'il faut refaire?



- **On remarque que dans le graphe de dépendances ci dessus, chaque action a pour entrée et sortie un fichier**
  - Si la date de modification t_e du fichier d'entrée est supérieure à la date de modification t_s du fichier de sortie (i.e. entrée plus récente que sortie)
    - Exécuter les actions correspondantes et celles qui en découlent
  - Si le fichier de sortie n'existe pas, on applique l'action
  - Si le fichier d'entrée n'existe pas, make cherche à le produire en trouvant une règle dont il est la sortie

Plan

1. *Généralités :*
   - *Rappels: compilation séparée*
   - *Objectifs*

☞ l'outil make

- *cible, dépendance*
- *fichier Makefile de base*
- *Makefiles et variables*
- *Makefile et règles implicites*
- *Structuration des Makefile*
- *un exemple plus sophistiqué pour la gestion des dépendances au fichiers .h*

TELECOM ParisTech

Outil make, présentation général

- **Outil extrêmement simple dans un usage basique, et potentiellement très sophistiqué dans un usage avancé**

- **3 concepts seulement pour pouvoir démarrer**
  - Cible, dépendance, commandes

- **Des possibilités très variées**
  - A noter que make n'est pas dédié au langage C (il est notamment très utilisé pour la production de fichier PDF à partir de fichiers Latex)

TELECOM ParisTech

Cible, dépendances, commandes

- Un fichier `Makefile` contient la représentation du graphe de dépendance d'une application sous **forme de texte.**
- Chaque lien de dépendance est défini via une ***règle*** :
  `cible : dépendance 1 ... dépendance N`
- Il contient aussi, à la suite de chaque règle de dépendance, les actions à entreprendre pour passer des dépendances à la cible.
- Un fichier `Makefile` est donc une succession de lignes de dépendance et de lignes d'action :

ligne de dépendance	`cible : dépendance1 … dépendanceN`
ligne d'action	commande(s) à exécuter pour maintenir cible si une `dépendanceI` est modifiée

TELECOM ParisTech

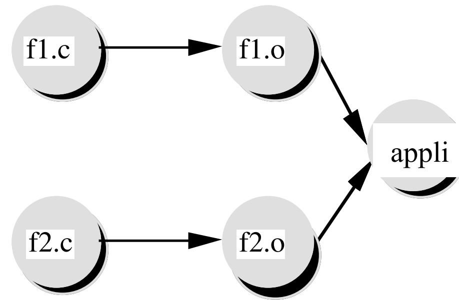
Plan

1. *Généralités :*
    - *Rappels: compilation séparée*
    - *Objectifs*

2. l'outil make
    - *cible, dépendance*
    - ☞ *fichier Makefile de base*
    - *Makefiles et variables*
    - *Makefile et règles implicites*
    - *Structuration des Makefile*
    - *un exemple plus sophistiqué pour la gestion des dépendances au fichiers .h*

TELECOM ParisTech

Exemple simpliste de fichier Makefile



- Voici un fichier Makefile qui effectue le travail représenté sur le schéma précédent. *(<TAB> indique qu'il FAUT une tabulation en début de ligne d'action) :*

```
########### appli dépend de f1.o et f2.o:
appli : f1.o f2.o
<TAB>  gcc f1.o f2.o  -o appli

############ f1.o dépend de f1.c
f1.o : f1.c
<TAB> gcc -c -Wall f1.c

########### f2.o dépend de f2.c
f2.o : f2.c
<TAB> gcc -c -Wall f2.c
```

- Note: # permet de mettre des commentaires dans le Makefile (comme // en C)

TELECOM ParisTech

Execution de *make*

- Après une modification de l'un des fichiers impliqués dans le Makefile, il suffit de donner la commande :

```
make appli
ou
make
```

- make sans argument supplémentaire traite la première cible
- make avec un argument supplémentaire traite la cible donnée en argument
- **Pour une cible donnée, make parcourt récursivement les dépendances en fonction des dates de modification des fichiers correspondants**:
  - si une cible est plus ancienne qu’une dépendance, les actions pour reconstruire la cible sont effectuées.
  - si la cible ou la dépendance ne correspond pas à un fichier, make effectue les commandes pour reconstruire la cible

Execution de make et code de retour des applications

- **Lorsque make exécute une commande, par exemple gcc, il sait si l'exécution de gcc a réussi ou échoué grâce au « return » de l'application gcc.**

- **Convention: 0 signifie que tout s'est bien passé**

- **En cas de valeur différente de 0, make considère que l'exécution a échoué**
  - Par défaut, make s'arrête
  - Avec l'option –i, il ignore les erreurs et continue

TELECOM ParisTech

Make est évidemment plus sophistiqué

- **Rappelons que nous avons décidé de « programmer » la production du logiciel.**

- **Le langage des Makefile permet également de:**
  - Définir des variables
  - Définir des règles de substitution
  - Utiliser des règles implicites et variables prédéfinies
  - Structurer les Makefile en différents fichiers inclus les uns à partir des autres
  - Redéfinir des règles existantes
  - …

TELECOM ParisTech

Make et répétition de cibles

- **En répétant une cible, on peut vouloir**
  - la surcharger (le dernier qui a parlé a raison)

```
f1.o: f1.c
	gcc –c f1.c

f1.o: f1.c
	gcc –c –g f1.c
```

  - la compléter (par concaténation des commandes)

```
f1.o: f1.c
	gcc –c f1.c

f1.o:: f1.c
	ls -lrt f1.c
```


TELECOM ParisTech

1. *Généralités :*
   - *Rappels: compilation séparée*
   - *Objectifs*

2. l'outil make
   - *cible, dépendance*
   - *fichier Makefile de base*
   - ☞ *Makefiles et variables*
   - *Makefile et règles implicites*
   - *Structuration des Makefile*
   - *un exemple plus sophistiqué pour la gestion des dépendances au fichiers .h*

Makefile et variables (macros)

- **Permet de définir des règles de substitution de texte**

- **Syntaxe:**
  - Déclaration d'une variable

    EXEC = /home/borde/zoom

  - Utilisation d'une variable

    $(EXEC)

A l'exécution de make, $(EXEC) prend la valeur /home/borde/zoom

- **Pratique: on peut aussi utiliser des variables d'environnement ($(HOME), etc…)**

Les variables de makefile pour le C

Exemple de variables communes pour produire un exécutable à partir de code C :

Rôle	Nom	Exemple d'initialisation
option de compilation	`CFLAGS`	`CFLAGS=-c -g -Wall`
option d'éd. de liens	`LDFLAGS`	`LDFLAGS= -g -lm`
fichiers objets	`OFILES`	`OFILES= f1.o f2.o`
fichiers sources	`CFILES`	`CFILES= f1.c f2.c`
nom du compilateur	`CC`	`CC= gcc`
nom de l'éd. de liens	`LD`	`LD= gcc`
commande rm	`RM`	`RM= /bin/rm`
nom du programme	`PROG`	`PROG= appli`

la cible n'est pas forcément un fichier, ici `make clean` va nettoyer le répertoire courant :

```
...
clean :
	$(RM) $(OFILES)
...
```


TELECOM ParisTech

Exemple de Makefile avec variables

Nouvelle version, paramétrée, du fichier Makefile :

```
######################################
BINDIR    =     /usr/local/bin
CFLAGS    =     -c -g -Wall
LDFLAGS   =     -g -lm
OFILES    =     f1.o f2.o
CC        =     $(BINDIR)/gcc
LD        =     $(BINDIR)/gcc
RM        =     /bin/rm -f
PROG      =     appli

######################################
appli: $(OFILES)
     $(LD) $(LDFLAGS) $(OFILES) -o $(PROG)

f1.o: f1.c
     $(CC) $(CFLAGS) f1.c

f2.o: f2.c
     $(CC) $(CFLAGS) f2.c

######################################
clean:
     $(RM) $(OFILES) core
```

A quoi servent ces variables… ?

TELECOM ParisTech

Exemple de Makefile avec variables

Nouvelle version, paramétrée, du fichier Makefile :

```
#######################################
BINDIR    =    /usr/local/bin
CFLAGS    =    -c -g -Wall
LDFLAGS   =    -g -lm
OFILES    =    f1.o f2.o
CC        =    $(BINDIR)/gcc
LD        =    $(BINDIR)/gcc
RM        =    /bin/rm -f
PROG      =    appli

#######################################
appli: $(OFILES)
	$(LD) $(LDFLAGS) $(OFILES) -o $(PROG)

f1.o: f1.c
	$(CC) $(CFLAGS) f1.c

f2.o: f2.c
	$(CC) $(CFLAGS) f2.c

#######################################
clean:
	$(RM) $(OFILES) core

#######################################
run: appli
	./appli
```

A quoi servent ces variables… ?

-g: on peut l'ajouter à toutes les étapes de compilation en l'ajoutant une seule fois dans le Makefile

De même si on veut changer de version de compilateur…

TELECOM ParisTech

Maintenant qu'on a des cibles plus variée, on peut…

Enchainer des constructions de cibles:

- make clean run

→ Make va enchainer séquentiellement les cibles clean, run
→ Comme run dépend de appli, make va tout recompiler et lancer l'exécutable

- make run

→ Ici, seules les étapes de compilation nécessaires sont effectuées

TELECOM ParisTech

Makefile et modifications de variables (macros)

- **On peut définir des variables dans l'exécution de make**

```
make CC=gcc all
```

- **On peut concaténer des variable, et conditionner l'inclusion de « lignes de code »:**

```
ifdef DEBUG
CFLAGS += -g
endif
```

S'utilise comme suit:

```
make DEBUG=1
```

- **On peut définir des règles de substitution de motif:**

```
$(Variable:srcprefix%srcsuffix=tgtprefix%tgtsuffix)
```

Exemple: `NEW_PATH = $(OLD_PATH:%txt=/tmp/%csv)`

Plan

1*. Généralités :*

- *Rappels: compilation séparée*
- *Objectifs*

2. l'outil make

- *cible, dépendance*
- *fichier Makefile de base*
- *Makefiles et variables*
- ☞ *Makefile et règles implicites*
- *Structuration des Makefile*
- *un exemple plus sophistiqué pour la gestion des dépendances au fichiers .h*

Règles implicites

- **Si make rencontre une cible (via une dépendance par exemple) pour laquelle aucune cible n'est définie, il utilise une règle implicite**

```
appli: f1.o f2.o
	gcc f1.o f2.o –o appli
```

Si la règle de construction de main.o n'est pas définie, make utilise une règle implicite:

```
%.o : %.c
	$(CC) $(CPPFLAGS) $(CFLAGS) –c $<
```

Règle avec patron (pattern) et variable automatique

Où CC, etc. sont des variables (potentiellement prédéfinies). Par contre, $< est une variable automatique car elle est calculée à l'exécution en fonction de la règle. $< vaut « main.c » si la cible est « main.o »

Makefile et variables automatiques

- **On peut aussi référencer les cibles, ou dépendances via des variables (macros) automatiques**

`$@`	La liste des cibles.
`$^`	La liste des dépendances
`$<`	La première dépendance

http://www.gnu.org/software/make/manual/make.html#Automatic-Variables

Plan

1. *Généralités :*
    - *Rappels: compilation séparée*
    - *Objectifs*

2. l'outil make
    - *cible, dépendance*
    - *fichier Makefile de base*
    - *Makefiles et variables*
    - *Makefile et règles implicites*
    - ☞ *Structuration des Makefile*
    - *un exemple plus sophistiqué pour la gestion des dépendances au fichiers .h*

Inclusion de Makefile

- **On peut inclure un fichier Makefile à partir d'un autre via l'utilisation du mot clé**

include

- **Les fichiers include auxiliaires ont souvent une extension .mk**

include setup.mk

- **L'option –I de make permet d'indiquer à l'outil la liste des répertoires où aller chercher des Makefile inclus**

- **Remplacer include par –include permet de dire que le fichier à inclure est optionnel (make s'exécutera normalement même si le fichier n'existe pas)**
  - Permet notamment d'inclure des fichiers auxiliaires produits par des règles du même Makefile (voir exemple sophistiqué plus bas)

Décomposition de Makefile

- **Pratique: décomposer les Makefile pour produire une application en suivant la décomposition de ses sources en sous-répertoires**

- **Option –C (pour change directory)**

```
all:
	make –C math all
	make –C graphs all
	make –C core all
	make –C ui all
```


TELECOM ParisTech

Plan

1. *Généralités :*
   - *Rappels: compilation séparée*
   - *Objectifs*

2. l'outil make
   - *cible, dépendance*
   - *fichier Makefile de base*
   - *Makefiles et variables*
   - *Makefile et règles implicites*
   - *Structuration des Makefile*

   ☞ *un exemple plus sophistiqué pour la gestion des dépendances au fichiers .h*

Dépendances et fichiers .h

- **On suppose que f2.c contient la directive au pré-processeur: #include "f2.h"**

- **Que faire si f2.h est modifié? …**
  - Refaire les cibles impactées « une par une »? ***NON***
  - Make s'en débrouille tout seul? ***PAS VRAIMENT***

TELECOM ParisTech

Exemple de *makefile*

Première partie

```
#  Cree l'executable 'myprog' a partir des fichiers 'personne.c' et 'main.c'
#  Auteur: Reda Dehak

# Nom de l'executable a creer
PROG= myprog

# Fichiers source (NE PAS METTRE les .h ni les .o,
SEULEMENT les .c)
SOURCES= f1.c f2.c

# Fichiers objets (ne pas modifier, sauf si l'extension n'est pas .c)
OBJETS=${SOURCES:%.c=%.o}

# Compilateur C
CC= gcc
# Options du compilateur C
# -g permet de debugger, -O optimise, -Wall affiche les erreurs
CFLAGS= -g -Wall

# Editeur de lien
LD=gcc

# Options de l'editeur de liens
LDFLAGS=

# Bibliothèques a utiliser
# Exemple pour Qt: LDLIBS = -L/usr/local/qt/lib -lqt
LDLIBS=
```

Options de gcc:
-L → où aller chercher les librairies

-lXYZ → l'archive de la bibliothèque s'appelle libXYZ.a

TELECOM ParisTech

Deuxième partie

```
###############################################
# Regles de construction/destruction des .o et de l'executable
# (depend sera un fichier contenant les dependances)

all: ${PROG}

${PROG}: depend ${OBJETS}
		${LD} -o ${PROG} ${LDFLAGS} ${OBJETS} ${LDLIBS}

clean:
		$(RM) $(PROG) *.o depend

# Gestion des dependances : creation automatique des dependances en utilisant
# l'option -MM de gcc (attention certains compilateurs n'ont pas cette option)

depend:
		${CC} -MM ${SOURCES} > depend

###############################################
# Regles implicites

.SUFFIXES: .c
.c.o:
		$(CC) -c $(CFLAGS) -o $@ $<

###############################################
# Inclusion du fichier des dependances
-include depend
```

A illustrer

TELECOM ParisTech

Exercice: organisation de l’application

Lorsque les fichiers constituant une application deviennent nombreux, il est souhaitable de les organiser en plusieurs répertoires.

On peut vouloir séparer les fichiers sources et objets des fichiers `include`. On pourrait adopter l'organisation suivante :

Type de fichiers	Répertoire de stockage
sources et objets correspondants	`src`
`"include"`	`hdr`

La seule modification à faire est d'indiquer à `gcc` de chercher les fichiers de type .h dans le répertoire `hdr` en utilisant l'option `-I` .
<u>On ne modifie pas les sources.</u>

```
CFLAGS= -c -g -Wall -I../hdr
```


TELECOM ParisTech

Exercice sur les makefile

Ecrire un fichier Makefile, qui:

- Via la cible appli, automatise la production d'un exécutable appli à partir de fichiers sources main.c, foncs.c, et fonc.h. foncs.h contient la signature des fonctions implémentées dans foncs.c et main.c utilise des fonctions implémentées dans foncs.c.
- Utilise la compilation séparée pour accélérer la production du binaire exécutable en cas de modifications
- Gère les dépendances entre fichiers
- Via la cible clean, supprime l'ensemble des fichier binaires produits

Avec votre solution, que se passe-t-il si on tape « make clean »? Et si on tape « make »?

TELECOM ParisTech

Au dela de la chaine de production

- **Les activités de développement logiciels ne se limitent pas au « codage »**
  - Documentation
  - Test (unitaire/fonctionnels/intégration/non-régression)
  - Déploiement
  - Maintenance (corrective/évolutive)

- **Beaucoup d’autres outils essentiels**
  - Gestion de version (svn/git)
  - Intégration continue (jenkins)
  - Suites de test

Un outil essentiel au développement en C: gdb

Les programmes ne fonctionnent généralement pas correctement dès la première exécution. Lors de la mise au point d'une application volumineuse, la trace avec `printf` s'avère vite fastidieuse et inefficace.

Pour une mise au point efficace, on utilise l'option "debug" qui permet d'exécuter un programme en mode pas à pas, de visualiser le contenu des variables, leur adresse, etc.

Cette option "debug" (option `-g`) doit être indiquée à la compilation et à l'édition de liens :

```
gcc -g -c -Wall f1.c
gcc -g -c -Wall f2.c
gcc -g f1.o f2.o -lm
```

Ensuite on pourra utiliser un debugger type `gdb`

Quelques commandes de base

- `gdb ./a.out` pour lancer gdb sur l'exécutable produit via les commandes ci-dessus
- `start argv1` pour lancer lancer le programme avec un argument arg1 passé à main
- `b f1.c : 24` pour placer un breakpoint à la ligne 24 du fichier f1.c
- `c` pour continuer l'exécution jusqu'au prochain breakpoint
- `bt` pour afficher le contexte d'exécution (séquence d'appels) ayant abouti au breakpoint
- `p` var pour inspecter le contenu de la variable var dans le contexte d'exécution du breakpoint