Outil de développement : (gnu)make

Brigitte Wrobel-Dautcourt

Novembre 2007

Nancy-Université
Université Henri Poincaré

Département Informatique - Faculté des Sciences
Université Henri Poincaré - Nancy 1

Versions précédentes : février 97, septembre 97, octobre 98, janvier 99, octobre 99, février 2000, février 2001, septembre 2005, novembre 2006

Bibliographie

Managing Projects with GNU Make, Third Edition. Robert Mecklenburg. O'Reilly, 2004.

Software Portability with imake, Second Edition, Paul Dubois. 1996.

---

1. Introduction

Outil de développement : (gnu)make
└Introduction

Introduction

make est une commande Unix.
make est un générateur de commandes.

- À partir d'un fichier de description, make crée une séquence de commandes exécutées par le shell de Unix.
- Généralement les commandes générées par make gèrent la maintenance d'une application logicielle :
  - construction de la version finale exécutable d'un ensemble de programmes (programmation en C, en assembleur, en C++, ...)
  - installation de logiciels
  - nettoyage des fichiers temporaires et déplacement de fichiers
  - mise-à-jour de bibliothèques
  - formatage de documents, ...

2 / 57

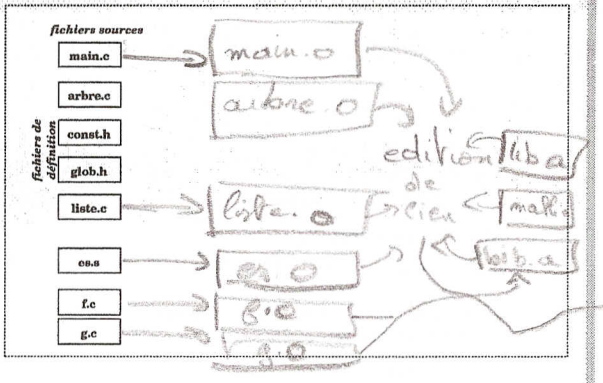
Outil de développement : (gnu)make
└Introduction

Motivations

- simplifier le déclenchement des suites d'opérations souvent répétées sur des fichiers
- optimiser la reconstruction d'exécutables et de bibliothèques dépendant de fichiers modifiés

3 / 57

Outil de développement : (gnu)make
└Introduction



4 / 57

Outil de développement : (gnu)make
└Introduction

Il faut deux commandes pour construire l'exécutable **appli** (sans tenir compte de la construction de `bib.a`) :

```
as es.s -o es.o
gcc main.c arbre.c liste.c es.o -o appli -lm bib.a
```

fichiers construits : `es.o`
`appli`

Quand un seul fichier source en langage C change, la compilation de **tous** les sources doit s'effectuer :

- rapide à écrire → TRÈS COÛTEUX
- long à s'exécuter

On voudrait :

- rapide à écrire
- rapide à s'exécuter

5 / 57

Rappel de quelques options du compilateur C gcc :

- **-c** : construction seule du fichier objet,
- **-o** : choix du nom de l'exécutable,
- **-lm** : utilisation de la bibliothèque mathématique.

La commande **as** réalise l'appel à l'assembleur.

Outil de développement : (gnu)make
└Introduction

Compilation séparée

Reconstruction séparée et indépendante de chaque fichier objet.

Liste complète des commandes pour construire l'exécutable `appli` *et les différents fichiers objets nécessaires :*

```
gcc -c main.c
gcc -c arbre.c
gcc -c liste.c
as es.s -o es.o
gcc -c f.c
gcc -c g.c
ar cur bib.a f.o g.o

gcc main.o arbre.o liste.o es.o -o appli -lm lib.a
```

6 / 57

Pour simplifier l'écriture des commandes, nous avons volontairement enlever les options **-ansi**, **-Wall** et **-g** lors de l'appel du compilateur **gcc**, elles seront ultérieurement ajoutées dans le fichier **Makefile** (page 15).

La commande **ar** permet de construire une bibliothèque (ensemble de fonctions), de nom `bib.a` pour l'exemple donné, à partir de fichiers objets (`f.o` et `g.o` dans l'exemple).

Outil de développement : (gnu)make
└Introduction

Complément sur les bibliothèques :

- **librairies statiques :**
  - extension **.a**
  - simples archives de fichiers **.o** (commande **ar**)
  - le code est inséré dans l'exécutable à la compilation
- **librairies dynamiques :**
  - extension **.so** (*share object*), **dynlib**, ...
  - ajout de l'option **shared** à la compilation par **gcc**
  - le code n'est pas inséré dans l'exécutable
  - il est chargé dynamiquement à l'exécution

7 / 57

Outil de développement : (gnu)make
└Introduction

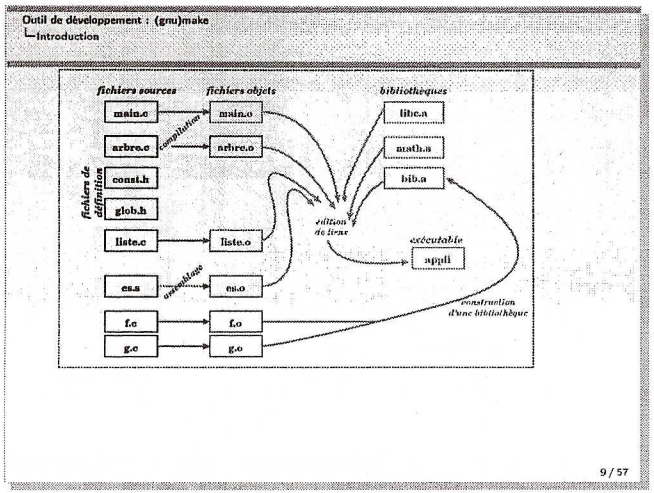
Complément sur les bibliothèques (suite) :

- **avantages :**
  - les programmes exécutables sont beaucoup plus petits
  - moins de *swap* à l'exécution car le code est partagé
- **inconvénients :**
  - la librairie doit exister à l'exécution (éventuel problèmes de licene...)
  - et il faut trouver la bonne...

Les librairies dynamiques sont recherchées dans la variable d'environnement :

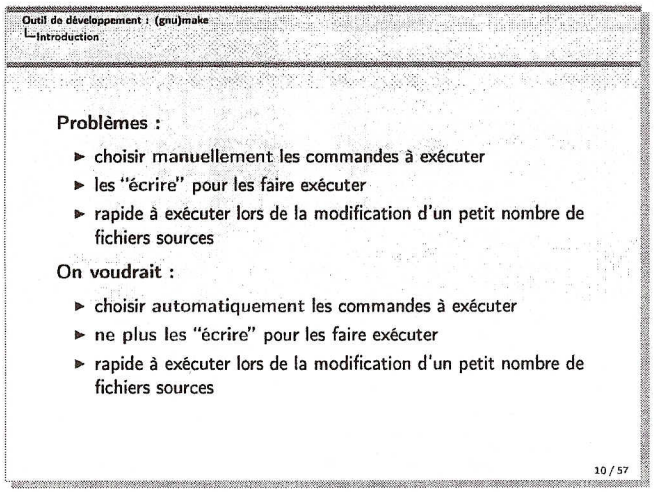
LD_LIBRARY_PATH

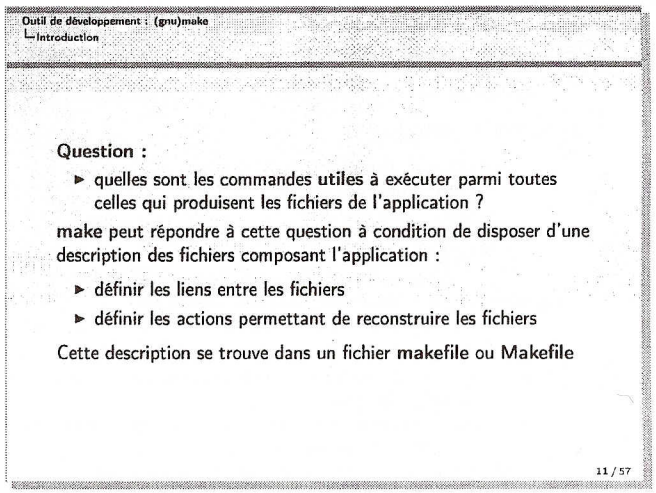
8 / 57



Si on modifie un ou plusieurs fichiers sources ou un ou plusieurs fichiers de définition, il est nécessaire de faire à nouveau l'édition de liens après avoir recompilé les fichiers sources modifiés ou incluant des fichiers de définition modifiés.

Cette reconstruction partielle est normalement répétée un nombre très important de fois au cours de la construction d'un logiciel.



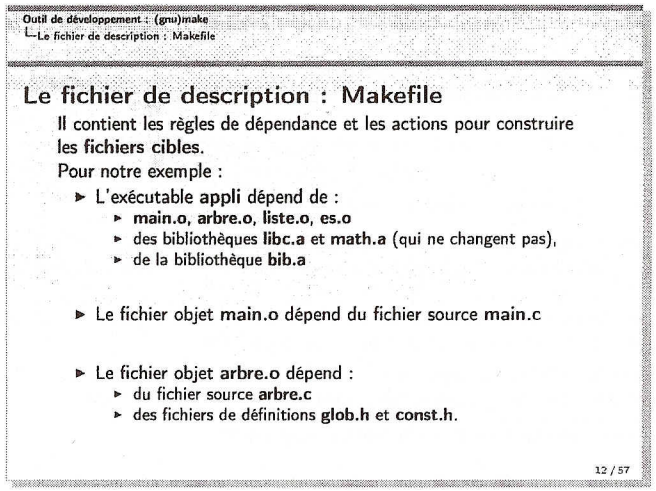


make permet de sélectionner automatiquement les seules commandes utiles de mise-à-jour.

De plus, le fichier de description utilisé par ***make*** demande d'écrire explicitement les dépendances entre les fichiers.

Ceci contribue à garder la maîtrise sur un logiciel de taille importante.

2. Le fichier de description : Makefile



3. Syntaxe d'une règle

Outil de développement : (gnu)make
└Syntaxe d'une règle

Syntaxe d'une règle

```
cible : fich_a fich_b fich_c ...fich_q
tab     action_1
tab     action_2
tab     ...
tab     action_n
```

si (date(fich_i) > date(cible)) ou (cible n'existe pas)
alors faire séquentiellement les actions (action_1, ... action_n)

- make reconnaît une cible car elle est suivi de ":"
- make reconnaît une action car elle commence par le caractère de tabulation noté tab
- chaque fichier apparaissant dans la liste des dépendances est à son tour considéré comme une cible
- make procède en *profondeur d'abord* dans l'arbre des dépendances.

13 / 57

Le fichier de description ***Makefile*** est créé dans le répertoire contenant les fichiers sources.

Une cible doit obligatoirement être le nom d'un fichier[1] puisque ***make*** va chercher la date de ce fichier pour la comparer avec les dates des fichiers apparaissant dans la liste des fichiers dont il dépend : `fich_a`, `fich_b` ... `fich_q`.

Une seule règle au maximum peut être associée par cible ; c'est-à-dire qu'un fichier cible n'apparaît qu'une seule fois dans le fichier de description.

À chaque règle correspond un certain nombre d'actions, conduisant généralement à la construction du fichier cible.

Une action est une commande interprétée par le shell courant, elle est généralement l'appel à un compilateur ou à un assembleur (`gcc`, `as`, ...), mais elle peut également être :

- un appel à n'importe quelle commande du système d'exploitation :
  `ls`, `cd`, `rm`, `echo`, `cp`, ...
- un appel à l'un de vos shell-scripts (écrit en `sh`, `csh`, `tcsh`, ...),
- l'appel à une commande aliasée à condition de respecter la syntaxe et les contraintes du shell qui interprète les actions définies dans le fichier ***Makefile*** (voir la variable SHELL définie page 17).

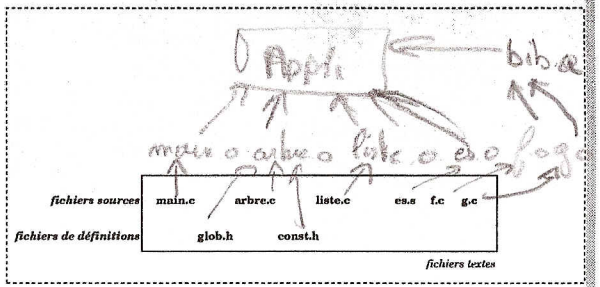
Les règles de dépendances forment un arbre dont on "exécutera" les feuilles d'abord pour remonter progressivement vers la racine, c'est-à-dire la première cible.

Le fichier ***Makefile*** contient explicitement toutes les règles de dépendances mais ***make*** peut déterminer certaines règles par lui-même avec l'utilisation des règles de suffixes (page 23).

[1] sauf dans certains cas particuliers, voir l'exemple donné page 12.

Outil de développement : (gnu)make
└Syntaxe d'une règle

Arbre de reconstruction



14 / 57

Tous les fichiers `*.c`, `*.s` et `*.h` de cet exemple sont susceptibles d'être modifiés, aussi ils doivent figurer dans les dépendances définissant les fichiers cibles.

Pour notre exemple, la bibliothèque `bib.a` peut également être modifiée, il est donc important (disons même indispensable) de la faire apparaître dans l'arbre des dépendances et dans le fichier ***Makefile***.

En revanche, les bibliothèques standards, ainsi que leurs fichiers de définitions associés (par exemple : `stdio.h`, `math.h`, ...) sont considérés comme stables et n'ont donc pas besoin de figurer ni dans l'arbre de dépendances, ni dans le fichier ***Makefile***.

Outil de développement : (gnu)make
└Syntaxe d'une règle

Fichier Makefile

```
appli : main.o arbre.o liste.o es.o bib.a
        gcc arbre.o liste.o main.o es.o -o appli -lm bib.a

main.o : main.c
        gcc -c main.c

arbre.o : arbre.c glob.h const.h
        gcc -c arbre.c

liste.o : liste.c
        gcc -c liste.c

es.o : es.s
        as es.s -o es.o

bib.a : f.o g.o
        ar cur bib.a f.o g.o
f.o : f.c
        gcc -c f.c
g.o : g.c
        gcc -c g.c
```

15 / 57

Outil de développement : (gnu)make
└Syntaxe d'une règle

Syntaxe (suite)

- un commentaire est précédé du caractère #

```
# construction de l'application appli
appli : main.o arbre.o liste.o es.o bib.a
        gcc arbre.o liste.o main.o es.o -o appli -lm bib.a

# compilation du fichier C main.c
main.o : main.c
        gcc -c main.c
...
```

- mettre \ juste avant le *return* quand une ligne se continue sur la ligne suivante (pour rendre le code lisible)

```
appli : main.o arbre.o liste.o es.o bib.a
        gcc arbre.o liste.o main.o es.o -o appli \
            -lm bib.a
```

16 / 57

Compléments :

- Désactivation de l'écho des lignes de commandes lors de leur exécution :

```
@commande
```

Une ligne de commande précédée par @ n'est pas affichée lors de son exécution, seul le résultat de son exécution est affiché, ceci est utile quand la commande provoque également un affichage ; par exemple :

```
liste.o : liste.c
        gcc -c liste.c
        mv liste.o ../objets_commun/
        @ echo déplacement de liste.o effectué
```

- Poursuivre l'exécution de la commande ***make*** même si une commande se termine sur un code d'erreur (c'est-à-dire quand une valeur non nulle a été retournée dans la variable d'environnement ***status***) :

```
-commande
```


Outil de développement : (gnu)make
└Syntaxe d'une règle

Par défaut :

```
bash-3.00$ gmake
```

exécute la première règle de dépendance écrite dans le fichier de description.
Plus généralement, on peut appeler make sur n'importe quelle cible indiquée dans le fichier de description, par exemple :

```
bash-3.00$ gmake arbre.o
```

17 / 57

gmake : On utilise la version ***gnu*** de ***make***.

Le projet GNU (prononcer "gnou" avec un g audible[1]) a débuté en 1984 sous l'initiative de l'américain Richard Stallman afin de créer un système d'exploitation similaire à UNIX utilisant du logiciel libre.

Il a également lancé la "Free Software Foundation" pour traiter les aspects juridiques et organisationnels du Projet GNU et pour répandre l'usage et la connaissance du Logiciel Libre. La Free Software Foundation formula la Licence Publique Générale GNU (LPG) et la Licence Publique Générale GNU Limitée, qui, au fil des ans, sont devenues les licences de Logiciel Libre les plus utilisées.

Le Projet GNU consiste en des sous-projets, plus petits, maintenus par des volontaires ou des sociétés et ayant généralement le but de créer ou de maintenir un composant fonctionnel. Ces sous-projets sont appelés "Projets GNU" ou "Projets Officiels GNU".

Le nom du Projet GNU provient d'un acronyme récursif "GNU is Not UNIX". UNIX étant à l'origine non seulement un certain type de système mais également un produit, ceci cherchait à symboliser le fait que le projet GNU a pour objectif de créer un système compatible mais pas identique à UNIX.

Le système GNU est (comme les autres systèmes d'exploitation UNIX) modulaire et aujourd'hui le système GNU avec un noyau Linux - nommé Système GNU/Linux - est largement utilisé et fournit la base des "Distributions Linux".

---

[1] d'où le nombre incalculable de gnous utilisés pour illustrer les documents...

Messages d'erreur

```
bash-3.00$ gmake appli
'appli' is uptodate.

bash-3.00$ gmake toto
gmake:  don't know how to make toto.  Stop.
```

make imprime chaque exécution de commande sur le terminal

-n affiche seulement les commandes sans les exécuter (ce qui est pratique en mode "debug" du fichier Makefile)

-s fait travailler make en mode silencieux... ce qui nous permet de ne pas savoir ce qu'il fait :=)

-d affiche les règles de dépendances choisies

- l'oubli d'une tabulation [tab] devant une action provoque une erreur :

```
Make:  Must be a separator on rule line 3.  Stop.
```

la commande :

```
bash-3.00$ cat -vte Makefile
```

permet de visualiser les caractères utilisés dans le fichier Makefile :

-v : fait apparaître tous les caractères non-imprimables, sauf les tabulations, les fins de ligne, ...

-t : fait apparaître les tabulations par ^I (et les *form-feed* par ^L),

-e : fait apparaître les fins de ligne par $.

Les options **-e** et **-t** de la commande `cat` sont ignorées si l'option **-v** n'est pas spécifiée.


```
bash-3.00 $ cat Makefile
appli :  main.o arbre.o liste.o es.o bib.a
         gcc arbre.o liste.o main.o es.o -o appli \
         -lm -bib.a

main.o :  main.c
         gcc -c main.c
```

```
bash-3.00 $ cat -vte Makefile
appli :  main.o arbre.o liste.o es.o bib.a$
         gcc arbre.o liste.o main.o es.o -o appli \ $

^I-lm -bib.a$
$
main.o :  main.c$
^Igcc -c main.c$
$
```


La règle de dépendance peut être vide... mais dans ce cas, il faut que la cible ne corresponde pas à un nom de fichier.

Exemple :

```
nettoie :
        /bin/rm *.o core
```

make cherche le fichier `nettoie` dans le répertoire courant

ce fichier n'existe pas car on n'a aucune commande de création de ce fichier

→ **exécution systématique de l'action**

Attention :

Si vous créez un fichier de nom `nettoie` dans votre répertoire, les actions ne s'exécuteront plus car le fichier `nettoie`, ne dépendant de rien, sera toujours à jour...

Modifications du fichier Makefile

Pour notre exemple :

- on suppose que `es.s` et `es.o` se trouvent dans le répertoire `/users/assemb`
- on suppose que `f.c`, `g.c`, `f.o` et `g.o`, nécessaires à la construction de la bibliothèque `bib.a`, se trouvent dans le répertoire `/users/biblio/sources`
- on suppose que le fichier de bibliothèque `bib.a` se trouve dans le répertoire `/users/biblio`
- on ajoute les options `-g -ansi -Wall -pedantic` à la compilation


```
appli : main.o arbre.o liste.o /users/assemb/es.o /users/biblio/bib.a
        gcc -g -ansi -Wall -pedantic arbre.o liste.o main.o \
            /users/assemb/es.o -o appli -lm /users/biblio/bib.a

main.o : main.c
        gcc -g -ansi -Wall -pedantic -c main.c
arbre.o : arbre.c glob.h const.h
        gcc -g -ansi -Wall -pedantic -c arbre.c
liste.o : liste.c
        gcc -g -ansi -Wall -pedantic -c liste.c

/users/assemb/es.o : /users/assemb/es.s
        as /users/assemb/es.s -o /users/assemb/es.o

/users/biblio/bib.a : /users/biblio/sources/f.o \
        /users/biblio/sources/g.o
        ar cur /users/biblio/bib.a /users/biblio/sources/f.o \
            /users/biblio/sources/g.o
/users/biblio/sources/f.o : /users/biblio/sources/f.c
        gcc -g -ansi -Wall -pedantic -c /users/biblio/sources/f.c
/users/biblio/sources/g.o : /users/biblio/sources/g.c
        gcc -g -ansi -Wall -pedantic -c /users/biblio/sources/g.c
```


4. Définition de macros

Définition de macros

- pour simplifier l'écriture du fichier Makefile

Syntaxe :

```
NOM = chaîne de caractères
```

Utilisation :

`${NOM}` ou `$(NOM)`

- par convention, les noms des macros sont en MAJUSCULES
- substitution dans le texte du fichier **Makefile** de "NOM" par "chaîne de caractères"
- les définitions de macros sont à placer **en tête** du fichier **Makefile**

Une macro peut être utilisée :

- dans la définition d'une autre macro,
- dans le nom d'une cible,
- dans une règle de dépendance,
- dans une action.


```
CC = gcc
CFLAGS = -g -ansi -Wall -pedantic
OBJ = main.o arbre.o liste.o
BIB = /users/biblio/
SBIB = ${BIB}sources/
REP = /users/assemb/

appli : ${OBJ} ${REP}es.o ${BIB}bib.a
        ${CC} ${CFLAGS} ${OBJ} ${REP}es.o -o appli -lm ${BIB}bib.a
...
${REP}es.o : ${REP}es.s
        as ${REP}es.s -o ${REP}es.o
...
```

Fichier ***Makefile*** en entier :

```
CC = gcc
CFLAGS = -g -ansi -Wall -pedantic
OBJ = main.o arbre.o liste.o
BIB = /users/biblio/
SBIB = ${BIB}sources/
REP = /users/assemb/

appli : ${OBJ} ${REP}es.o ${BIB}bib.a
        ${CC} ${CFLAGS} ${OBJ} ${REP}es.o -o appli -lm ${BIB}bib.a

main.o : main.c
        ${CC} ${CFLAGS} -c main.c

arbre.o : arbre.c ${DEF}
        ${CC} ${CFLAGS} -c arbre.c

liste.o : liste.c
        ${CC} ${CFLAGS} -c liste.c

${REP}es.o : ${REP}es.s
        as ${REP}es.s -o ${REP}es.o

${BIB}bib.a : ${SBIB}f.o ${SBIB}g.o
        ar cur ${BIB}bib.a ${SBIB}f.o ${SBIB}g.o

${SBIB}f.o : ${SBIB}f.c
        ${CC} ${CFLAGS} -c ${SBIB}f.c

${SBIB}g.o : ${SBIB}g.c
        ${CC} ${CFLAGS} -c ${SBIB}g.c
```

Le message :

```
Bad character ( (octal 173), line3
```

indique généralement l'oubli du caractère $ lors de l'utilisation d'une macro.

Attention :

Il faut respecter les noms des objets en entier.

La règle définissant la cible `appli` indique que celle-ci dépend, entre autres, du fichier `${REP}es.o`. On doit donc trouver dans le fichier ***Makefile*** une règle permettant de construire la cible `${REP}es.o` et non la cible `es.o` qui n'existe pas (cette règle pourra être remplacée par une règle de suffixe - page 23).

Outil de développement : (gnu)make
└Définition de macros

Les variables globales d'environnement du shell sont considérées comme des macros, elles peuvent être utilisées dans un fichier de description Makefile.
Exemple : définition de la variable globale d'environnement DIR dans le shell courant (en bash) :

```
bash-3.00$ export DIR=/usr/proj
bash-3.00$ make prog
```

Fichier Makefile : *utilisation de la variable DIR, manipulée avec la syntaxe ${DIR}, dans la définition d'une autre macro et/ou dans une règle de dépendance et/ou dans une action ; par exemple :*

```
SRC = ${DIR}/src/
prog : ...
        gcc ${DIR}/main.c ...
```

Il n'y a pas de message d'erreur quand une macro utilisée n'est pas définie, elle est alors instanciée par la chaîne vide.

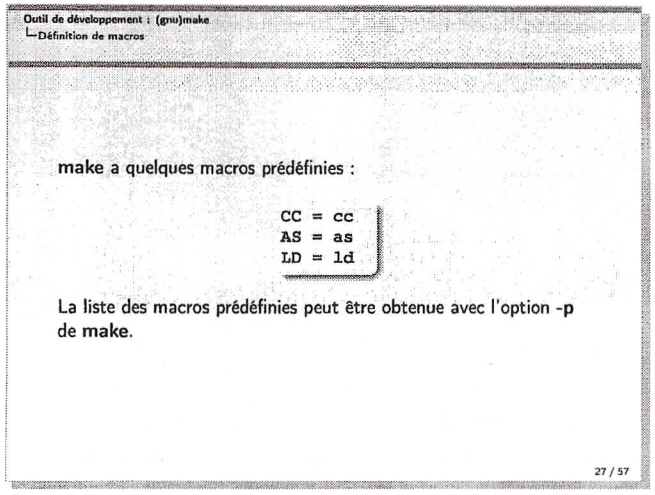
26 / 57

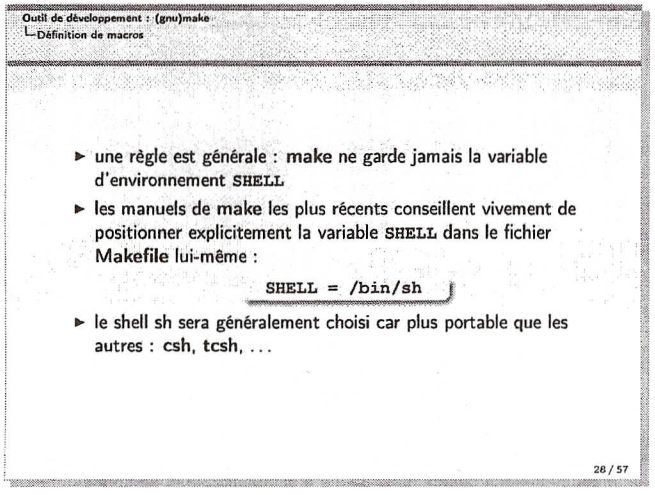
Attention, seules les variables globales d'environnement peuvent ainsi être utilisées comme des macros dans le fichier de description ***Makefile***.

Ainsi, le même fichier ***Makefile*** ne peut pas utiliser la macro `${DIR}`, si celle-ci a été définie dans le shell par l'instruction d'affectation d'une variable locale d'environnement (en csh pour l'exemple) :

```
bash-3.00$ set DIR = /essai/=ep
bash-3.00$ gmake prog
```

La variable `DIR` ne contient rien lors de l'exécution du fichier ***Makefile***.





make -p commence par afficher le contenu du fichier ***Makefile*** du répertoire courant, s'il existe, puis affiche les macros, les variables globales d'environnement (on y retrouve par exemple les variables `PATH`, `HOME`, ...) et les règles de suffixes (page 23).

Par défaut, la variable `SHELL` est fixée par ***make*** à une valeur soit codée en dur dans l'exécutable, soit mise dans un fichier de dépendances (cela varie avec les machines et les versions, il faut consulter chaque ***man***).

Une règle est générale : ***make*** ne garde jamais la variable d'environnement :

- sur Sun-OS4 et Solaris, le shell utilisé pour interpréter les actions définies dans le fichier ***Makefile***

est ***/bin/sh*** ;

l'exécution du fichier ***Makefile*** :

```
truc :
        @echo ${SHELL}
```

affiche : `/bin/sh`

Rappel sur le fonctionnement de ***make*** sur ce mini-fichier ***Makefile*** (expliqué précédemment page 12) :

> ***make*** cherche le fichier `truc` dans le répertoire courant. Ce fichier n'existe pas, aussi on exécute systématiquement les actions de la règle, soit ici l'affichage du contenu de la variable `SHELL`.
> Voir page 9 la signification du caractère `@`.

On a donc la possibilité d'utiliser des macros provenant de sources différentes, par défaut l'ordre de priorité est le suivant :

1. macros entrées sur la ligne de commande ***make***, si elles **suivent** l'appel de la commande ***make***

```
bash-3.00$ gmake appli DIR=/usr/proj
bash-3.00$ gmake appli "DIR =/usr/proj /usr/lib"
```

2. macros définies dans le fichier ***Makefile***
3. variables globales d'environnement du shell :

```
bash-3.00$ export DIR=/usr/proj
bash-3.00$ gmake appli
```

4. définitions par défaut de ***make***

Il est possible de modifier l'ordre de priorité des macros avec l'option ***-e*** de ***make***.

Soit par exemple, la variable globale d'environnement `TESTER`, définie d'une part dans le shell courant (en csh) :

```
bash-3.00$ setenv TESTER test_rep1
```

fichier ***Makefile*** :

```
TESTER = test_defaut

test : scrfile.c
       ${TESTER}srcfile.c
```

- Quand on exécute la commande :

```
bash-3.00$ gmake test
```

la macro `${TESTER}` utilisée est celle qui est définie dans le fichier **Makefile**, elle contient la valeur `test_defaut`.

- Quand on exécute la commande :

  bash-3.00$ `gmake -e test`

  l'ordre de priorité des macros est inversé et la macro `${TESTER}` utilisée est celle définie dans l'environnement courant ; elle contient la valeur `test_rep1`.

Ceci explique l'utilité de certaines définitions vides, une telle définition "efface" la définition précédente d'une variable d'environnement, par exemple la définition de la macro :

```
PROJECT_TREE =
```

dans un fichier ***Makefile***, permet d'oublier la valeur précédente de cette variable globale d'environnement.

Outil de développement : (gnu)make
└Définition de macros

Macros internes (variables automatiques)

Dans la plupart des cas, la cible à construire est le nom d'un fichier, aussi ce nom peut être remplacé par la macro @ (utilisée avec la syntaxe par $@ dans les actions), on peut alors modifier le fichier Makefile pour le rendre illisible...

$@ : représente la **cible courante** :

cible courante

```
es.o    : es.s
          ${AS} es.s -o   $@
```

cible es.o

29 / 57

Outil de développement : (gnu)make
└Définition de macros

$? : représente les fichiers de la liste de dépendance **plus récents** que la cible :

```
cible : fich1 fich2 ...  fichN
        commande         $?
```

1 ou n fichiers fich

30 / 57

Outil de développement : (gnu)make
└Définition de macros

Exemple 1 :

```
bib.a :   f.o g.o
          ar cur bib.a f.o g.o
```

- si `f.o` et `g.o` sont tous les deux plus récents que la cible `bib.a` :
  `ar cur bib.a f.o g.o`
- si seulement `f.o` est plus récent que la cible `bib.a` :
  `ar cur bib.a f.o`
- si seulement `g.o` est plus récent que la cible `bib.a` :
  `ar cur bib.a g.o`
- si aucun fichier n'est plus récent que la cible
  on ne fait rien, bien sûr !

31 / 57

Exemple 2 :

```
appli :   main.o liste.o ${REP}es.o ${BIB}bib.a
          ${CC} ${CFLAGS} $?  -o appli -lm ${BIB}bib.a
```

- si seulement `arbre.o` est plus récent que la cible `appli` :

```
gcc -g ...  arbre.o -o appli -lm /users/biblio/bib.a
```

FAUX

Exemple 3 :

```
main.o : main.c
        ${CC} ${CFLAGS} -c $?
```


$< : représente le premier nom de la liste de dépendance :

```
main.o : main.c
        ${CC} ${CFLAGS} -c  $<
                            main.c

arbre.o : arbre.c glob.h const.h
        ${CC} ${CFLAGS} -c  $<
                            arbre.c
```



```
SHELL = /bin/sh
CC = gcc
CFLAGS = -g -ansi -Wall
BIB = /users/biblio/
SBIB = ${BIB}sources/
REP = /users/assemb/

appli : main.o arbre.o liste.o ${REP}es.o ${BIB}bib.a
   ${CC} ${CFLAGS} main.o arbre.o liste.o ${REP}es.o -o $@ -lm \
      ${BIB}bib.a

main.o : main.c
   ${CC} ${CFLAGS} -c $<

arbre.o : arbre.c glob.h const.h
   ${CC} ${CFLAGS} -c $<

liste.o : liste.c
   ${CC} ${CFLAGS} -c $<

${REP}es.o : ${REP}es.s
   ${AS} $< -o $@

${BIB}bib.a : ${SBIB}f.o ${SBIB}g.o
   ar cur $@ $?

${SBIB}f.o : ${SBIB}f.c
   ${CC} ${CFLAGS} -c $<

${SBIB}g.o : ${SBIB}g.c
   ${CC} ${CFLAGS} -c $<
```

5. Règles implicites ou règles de suffixe

Zoom sur un extrait du fichier Makefile

```
main.o : main.c
   ${CC} ${CFLAGS} -c $<

arbre.o : arbre.c const.h glob.h
   ${CC} ${CFLAGS} -c $<

liste.o : liste.c
   ${CC} ${CFLAGS} -c $<

${BIB}f.o : ${BIB}f.c
   ${CC} ${CFLAGS} -c $<

${BIB}g.o : ${BIB}g.c
   ${CC} ${CFLAGS} -c $<
```


Règles de suffixe

- définition de deux règles de suffixes (règles implicites)

```
%.o : %.c
   ${CC} ${CFLAGS} -c $<

%.o : %.s
   ${AS} $< -o $@
```

- définition de la liste des suffixes :

```
.SUFFIXES : .o .c .s
```


Une règle de suffixe, aussi appelée règle implicite (par opposition aux règles explicites que nous venons de présenter) permet de définir les actions à effectuer pour créer un fichier avec un certain suffixe à partir d'un fichier comportant un autre suffixe.

La notation des règles de suffixe sous la forme :

```
.c.o:
        ${CC} ...
```


est obsolète...


```
SHELL = /bin/sh
CC = gcc
CFLAGS = -g -ansi -Wall
.SUFFIXES : .o .c .s    macro définissant les suffixes
BIB = /users/biblio/
SBIB = ${BIB}sources/
REP = /users/assemb/

appli : main.o arbre.o liste.o ${REP}es.o ${BIB}bib.a
     ${CC} ${CFLAGS} main.o arbre.o liste.o ${REP}es.o -o $@ \
     -lm ${BIB}bib.a
arbre.o : arbre.c const.h glob.h
     ${CC} ${CFLAGS} -c $<    inutile
${BIB}bib.a : ${SBIB}f.o ${SBIB}g.o
     ar cur $@ $?
%.o:%.c
     ${CC} ${CFLAGS} -c $<    } règles de suffixe
%.o:%.s
     as $< -o $@
```


On utilise une règle de suffixe seulement quand un fichier apparaissant dans une règle de dépendance ne figure pas explicitement comme cible dans le fichier ***Makefile***.

Remarques :

- l'ordre de définition des suffixes dans la liste donnée par `.SUFFIXES` est quelconque,
  - la ligne :

    ```
    .SUFFIXES: .k .j
    ```

    ajoute les suffixes `k` et `j` à la liste des suffixes déjà définis,
  - la ligne

    ```
    .SUFFIXES:
    ```

    détruit tous les suffixes précédemment reconnus,
  - la ligne

    ```
    .SUFFIXES:
    .SUFFIXES: .k .j
    ```

    remplace les suffixes précédemment reconnus par les suffixes `k` et `j` ;
- le fichier `arbre.o` dépend des deux fichiers de définition `const.h` et `glob.h`, il est donc indispensable d'écrire cette règle de dépendance dans le fichier ***Makefile*** ; en revanche, la règle de suffixe `.c.o` peut parfaitement s'appliquer, il est donc inutile de préciser l'action à faire pour reconstruire le fichier `arbre.o` à partir du fichier `arbre.c` ;
- il devient dans ce cas intéressant d'utiliser l'option `-d` lors de l'appel de `make` afin de tracer les règles de dépendance choisies.

5. Règles implicites ou règles de suffixe

Outil de développement : (gnu)make
└Règles implicites ou règles de suffixe

Zoom sur un extrait du fichier Makefile

```
main.o : main.c
   ${CC} ${CFLAGS} -c $<

arbre.o : arbre.c const.h glob.h
   ${CC} ${CFLAGS} -c $<

liste.o : liste.c
   ${CC} ${CFLAGS} -c $<

${BIB}f.o : ${BIB}f.c
   ${CC} ${CFLAGS} -c $<

${BIB}g.o : ${BIB}g.c
   ${CC} ${CFLAGS} -c $<
```

36 / 57

Outil de développement : (gnu)make
└Règles implicites ou règles de suffixe

Règles de suffixe

- définition de deux règles de suffixes (règles implicites)

```
%.o : %.c
   ${CC} ${CFLAGS} -c $<

%.o : %.s
   ${AS} $< -o $@
```

- définition de la liste des suffixes :

```
.SUFFIXES : .o .c .s
```

37 / 57

Une règle de suffixe, aussi appelée règle implicite (par opposition aux règles explicites que nous venons de présenter) permet de définir les actions à effectuer pour créer un fichier avec un certain suffixe à partir d'un fichier comportant un autre suffixe.

La notation des règles de suffixe sous la forme :

```
.c.o:
        ${CC} ...
```


est obsolète...

Outil de développement : (gnu)make
└Règles implicites ou règles de suffixe

```
SHELL = /bin/sh
CC = gcc
CFLAGS = -g -ansi -Wall
.SUFFIXES : .o .c .s    macro définissant les suffixes
BIB = /users/biblio/
SBIB = ${BIB}sources/
REP = /users/assemb/

appli : main.o arbre.o liste.o ${REP}es.o ${BIB}bib.a
     ${CC} ${CFLAGS} main.o arbre.o liste.o ${REP}es.o -o $@ \
     -lm ${BIB}bib.a
arbre.o : arbre.c const.h glob.h
     ${CC} ${CFLAGS} -c $<
${BIB}bib.a : ${SBIB}f.o ${SBIB}g.o
     ar cur $@ $?
%.o:%.c
     ${CC} ${CFLAGS} -c $<      } règles de suffixe
%.o:%.s
     as $< -o $@
```

38 / 57

On utilise une règle de suffixe seulement quand un fichier apparaissant dans une règle de dépendance ne figure pas explicitement comme cible dans le fichier ***Makefile***.

Remarques :

- l'ordre de définition des suffixes dans la liste donnée par `.SUFFIXES` est quelconque,
  - la ligne :

    ```
    .SUFFIXES: .k .j
    ```

    ajoute les suffixes `k` et `j` à la liste des suffixes déjà définis,
  - la ligne

    ```
    .SUFFIXES:
    ```

    détruit tous les suffixes précédemment reconnus,
  - la ligne

    ```
    .SUFFIXES:
    .SUFFIXES: .k .j
    ```

    remplace les suffixes précédemment reconnus par les suffixes `k` et `j` ;
- le fichier `arbre.o` dépend des deux fichiers de définition `const.h` et `glob.h`, il est donc indispensable d'écrire cette règle de dépendance dans le fichier ***Makefile*** ; en revanche, la règle de suffixe `.c.o` peut parfaitement s'appliquer, il est donc inutile de préciser l'action à faire pour reconstruire le fichier `arbre.o` à partir du fichier `arbre.c` ;
- il devient dans ce cas intéressant d'utiliser l'option `-d` lors de l'appel de `make` afin de tracer les règles de dépendance choisies.

Fonctionnement des règles de suffixes

```
appli : main.o arbre.o liste.o ${REP}es.o ${BIB}bib.a
   ${CC} ${CFLAGS} main.o arbre.o liste.o ${REP}es.o -o appli \
   -lm ${BIB}bib.a
```

S'il n'y a aucune règle définissant la cible `main.o`, make recherche dans le répertoire courant un fichier qui :

- soit plus récent que la cible à construire
- a le même nom que la cible avec un suffixe (`.x`) appartenant à la liste définie par `.SUFFIXES : .s .c .o`
- et pour lequel il existe une règle de suffixe de la forme : `%.o:%.x`

Ici, on trouve :

- un fichier `main.c` dans le répertoire courant
- un règle :

```
%.o:%.c
        ${CC} ${CFLAGS} -c $<
```

dépendance → `$<` → *fichier.c*

qui conduit à l'exécution de :

```
gcc -g -ansi -Wall -c main.c
```

pour construire le fichier `main.o`

make est "livré" avec :

- une liste de suffixes prédéfinis :

```
.SUFFIXES: .out .a .ln .o .c .cc .C .cpp .p .f .F .r .y .l .s .S
   .mod .sym .def .h .info .dvi .tex .texinfo .texi .txinfo .w
   .ch .web .sh .elc .el
```

- une liste de règles de suffixe :

```
%.o: %.c
   ${COMPILE.c) ${OUTPUT_OPTION) $<

%.o: %.s
   ${COMPILE.s) -o $@ $<
...
```

- une liste de variables (sur idefix) :

```
CC = cc
COMPILE.c = ${CC) ${CFLAGS) ${CPPFLAGS) ${TARGET_ARCH) -c
AS = as
COMPILE.s = ${AS) ${ASFLAGS) ${TARGET_MACH)
...
```


Il est possible de redéfinir les règles de suffixe déjà existantes.

Par ailleurs, il faut faire attention aux règles en compétition entre elles. Par exemple, si on veut construire la cible `prog` et que l'on dispose dans le même réeprtoire des deux fichiers `prog.c` et `prog.cc`, seule la première règle (la première trouvée) gagnera...

Il reste à définir les macros d'options pour adapter les macros déjà définies :

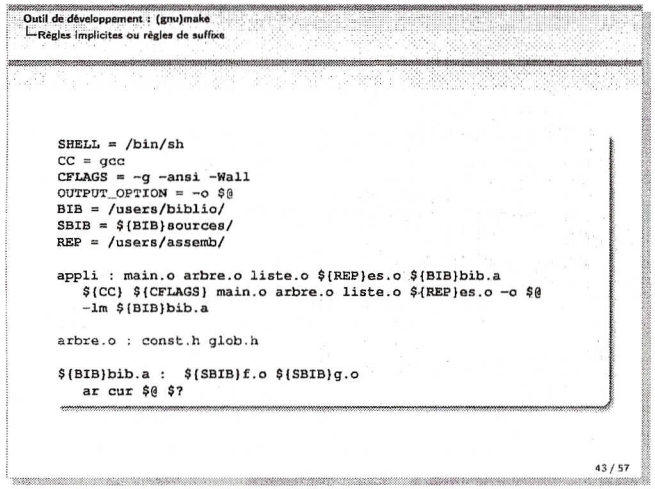
```
%.o: %.c
        ${COMPILE.c)                    ${OUTPUT_OPTION) $<
```

On veut obtenir :

```
%.o: %.c
   gcc -ansi -Wall -c -o $@ $<
```


Remarque :

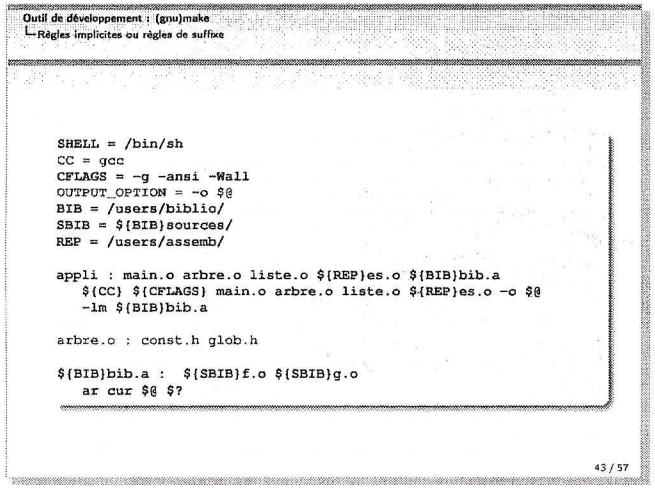
Les macros `OUTPUT_OPTION`, `CFLAGS` et `CPPFLAGS` ne sont pas définies par défaut, elles ne contiennent donc rien..., il n'est pas utile de leur donner une valeur pour cet exemple.



Extrait de la liste des règles des suffixes telle qu'elle est affichée par la commande **make -p** *sur la machine* **idefix**, *ici sur 2 colonnes :*

```
...
# Règles Implicites

%.out:

%.a:

%.ln:

%.o:

%:%.o
	$(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.c:

%: %.c
	$(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.ln: %.c
	$(LINT.c) -C$* $<

%.o: %.c
	$(COMPILE.c) $(OUTPUT_OPTION) $<

%.cc:

%: %.cc
	$(LINK.cc) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.cc
	$(COMPILE.cc) $(OUTPUT_OPTION) $<

%.C:

%: %.C
	$(LINK.C) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.C
	$(COMPILE.C) $(OUTPUT_OPTION) $<

%.cpp:

%: %.cpp
	$(LINK.cpp) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.cpp
	$(COMPILE.cpp) $(OUTPUT_OPTION) $<

%.p:

%: %.p
	$(LINK.p) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.p
	$(COMPILE.p) $(OUTPUT_OPTION) $<

%.f:

%: %.f
	$(LINK.f) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.f
	$(COMPILE.f) $(OUTPUT_OPTION) $<

%.F:

%: %.F
	$(LINK.F) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.F
	$(COMPILE.F) $(OUTPUT_OPTION) $<

%.f: %.F
	$(PREPROCESS.F) $(OUTPUT_OPTION) $<

%.r:

%: %.r
	$(LINK.r) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.r
	$(COMPILE.r) $(OUTPUT_OPTION) $<

%.f: %.r
	$(PREPROCESS.r) $(OUTPUT_OPTION) $<

%.y:

%.ln: %.y
	$(YACC.y) $<
	$(LINT.c) -C$* y.tab.c
	$(RM) y.tab.c

%.c: %.y
	$(YACC.y) $<
	mv -f y.tab.c $@

%.l:

%.ln: %.l
	@$(RM) $*.c
	$(LEX.l) $< > $*.c
	$(LINT.c) -i $*.c -o $@
	$(RM) $*.c

%.c: %.l
	@$(RM) $@
	$(LEX.l) $< > $@

%.r: %.l
	$(LEX.l) $< > $@
	mv -f lex.yy.r $@

%.s:

%: %.s
	$(LINK.s) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.s
	$(COMPILE.s) -o $@ $<

%.S:

%: %.S
	$(LINK.S) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.S
	$(COMPILE.S) -o $@ $<

%.s: %.S
	$(PREPROCESS.S) $< > $@

%.mod:

%: %.mod
	$(COMPILE.mod) -o $@ -e $@ $^

%.o: %.mod
	$(COMPILE.mod) -o $@ $<

%.sym:

%.def:

%.sym: %.def
	$(COMPILE.def) -o $@ $<

%.h:

%.info:

%.dvi:

%.tex:

%.dvi: %.tex
	$(TEX) $<

%.texinfo:

%.info: %.texinfo
	$(MAKEINFO) $(MAKEINFO_FLAGS) $< -o $@

%.dvi: %.texinfo
	$(TEXI2DVI) $(TEXI2DVI_FLAGS) $<

%.texi:

%.info: %.texi
	$(MAKEINFO) $(MAKEINFO_FLAGS) $< -o $@

%.dvi: %.texi
	$(TEXI2DVI) $(TEXI2DVI_FLAGS) $<
...
```


$% permet d'accéder au nom de la bibliothèque en cours de traitement.



Extrait de la liste des règles des suffixes telle qu'elle est affichée par la commande **make -p** *sur la machine* **idefix**, *ici sur 2 colonnes :*

```
...
# Régles Implicites

%.out:

%.a:

%.ln:

%.o:

%:%.o
	$(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.c:

%: %.c
	$(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.ln: %.c
	$(LINT.c) -C$* $<

%.o: %.c
	$(COMPILE.c) $(OUTPUT_OPTION) $<

%.cc:

%: %.cc
	$(LINK.cc) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.cc
	$(COMPILE.cc) $(OUTPUT_OPTION) $<

%.C:

%: %.C
	$(LINK.C) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.C
	$(COMPILE.C) $(OUTPUT_OPTION) $<

%.cpp:

%: %.cpp
	$(LINK.cpp) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.cpp
	$(COMPILE.cpp) $(OUTPUT_OPTION) $<

%.p:

%: %.p
	$(LINK.p) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.p
	$(COMPILE.p) $(OUTPUT_OPTION) $<

%.f:

%: %.f
	$(LINK.f) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.f
	$(COMPILE.f) $(OUTPUT_OPTION) $<

%.F:

%: %.F
	$(LINK.F) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.F
	$(COMPILE.F) $(OUTPUT_OPTION) $<

%.f: %.F
	$(PREPROCESS.F) $(OUTPUT_OPTION) $<

%.r:

%: %.r
	$(LINK.r) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.r
	$(COMPILE.r) $(OUTPUT_OPTION) $<

%.f: %.r
	$(PREPROCESS.r) $(OUTPUT_OPTION) $<

%.y:

%.ln: %.y
	$(YACC.y) $<
	$(LINT.c) -C$* y.tab.c
	$(RM) y.tab.c

%.c: %.y
	$(YACC.y) $<
	mv -f y.tab.c $@

%.l:

%.ln: %.l
	@$(RM) $*.c
	$(LEX.l) $< > $*.c
	$(LINT.c) -i $*.c -o $@
	$(RM) $*.c

%.c: %.l
	@$(RM) $@
	$(LEX.l) $< > $@

%.r: %.l
	$(LEX.l) $< > $@
	mv -f lex.yy.r $@

%.s:

%: %.s
	$(LINK.s) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.s
	$(COMPILE.s) -o $@ $<

%.S:

%: %.S
	$(LINK.S) $^ $(LOADLIBES) $(LDLIBS) -o $@

%.o: %.S
	$(COMPILE.S) -o $@ $<

%.s: %.S
	$(PREPROCESS.S) $< > $@

%.mod:

%: %.mod
	$(COMPILE.mod) -o $@ -e $@ $^

%.o: %.mod
	$(COMPILE.mod) -o $@ $<

%.sym:

%.def:

%.sym: %.def
	$(COMPILE.def) -o $@ $<

%.h:

%.info:

%.dvi:

%.tex:

%.dvi: %.tex
	$(TEX) $<

%.texinfo:

%.info: %.texinfo
	$(MAKEINFO) $(MAKEINFO_FLAGS) $< -o $@

%.dvi: %.texinfo
	$(TEXI2DVI) $(TEXI2DVI_FLAGS) $<

%.texi:

%.info: %.texi
	$(MAKEINFO) $(MAKEINFO_FLAGS) $< -o $@

%.dvi: %.texi
	$(TEXI2DVI) $(TEXI2DVI_FLAGS) $<
...
```

$% permet d'accéder au nom de la bibliothèque en cours de traitement.

Écriture de règles complémentaires

- **all** : généralement la première cible du fichier Makefile, elle regroupe dans les dépendances l'ensemble des exécutables à produire
- **clean**
  permet de détruire tous les fichiers intermédiaires créés par la résolution de la cible all
- **mrproper**
  supprime tout ce qui peut être régénéré et permet donc ensuite une reconstruction complète de l'application

(d'autres conventions de nommage de cibles dans le poly)

Quelques conventions de nommage de cibles pour un projet

- ***install*** : exécute la cible ***all***, puis copie l'exécutable, les bibliothèques, les données et les fichiers entête dans les répertoires de destination (installation)
- ***uninstall*** : détruit les fichiers créés lors de l' ***install***
- ***info*** : générer un fichier info
- ***man*** : générer les pages du manuel
- ***dist*** : crée un fichier tar de destination
- ***makedepend*** : calcule les dépendances et les ajoute au ***Makefile***

Makefile pour l'application utilisateur

```
SHELL = /bin/sh
CC = gcc
CFLAGS = -g -ansi -Wall
OUTPUT_OPTION = -o $@
BIB = /users/biblio/
SBIB = ${BIB}sources/
REP = /users/assemb/

all :  appli

appli : main.o arbre.o liste.o ${REP}es.o
   ${CC} ${CFLAGS} main.o arbre.o liste.o ${REP}es.o -o $@
   -lm ${BIB}bib.a

arbre.o : const.h glob.h

clean :
   -rm -f *.o

mrproper :  clean
   -rm -f appli
```


Makefile pour la construction de la bibliothèque

```
SHELL = /bin/sh
CC = gcc
CFLAGS = -g -ansi -Wall
OUTPUT_OPTION = -o $@
BIB = ../

all :  ${BIB}bib.a

${BIB}bib.a :  f.o g.o
   ar cur $@ $?

clean :
   -rm -f *.o

mrproper :  clean
   -rm -f ${BIB}bib.a
```

- **Construction automatique de la liste des fichiers sources présents dans un répertoire :**

```
SRC = $(wildcard *.c)
```

La notation seule ***.c** ne convient pas, il faut utiliser la commande **wildcard**

- **Génération de la liste des fichiers objets à partir de la liste des fichiers sources :**
  Construction d'une variable contenant la liste des fichiers où on remplace le suffixe **c** par le suffixe **o** :

```
OBJ = $(SRC:.c=.o)
```


Makefile de l'application utilisateur :

```
...
SRC = $(wildcard *.c)
OBJ = $(SRC:.c=.o)

all :  appli

appli : $(OBJ) $(REP)es.o $(BIB)bib.a
   $(CC) $(CFLAGS)$(OBJ) $(REP)es.o -o $@
   -lm $(BIB)bib.a

arbre.o : const.h glob.h

clean :
   -rm -f *.o

mrproper :  clean
   -rm -f appli
```


Makefiles conditionnels

```
...
DEBUG = oui

ifeq ($(DEBUG), oui)
   CFLAGS = -g -ansi -Wall
else
   CFLAGS = -ansi -Wall
endif
```


Makefile & sous-Makefiles

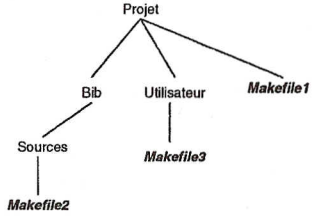
Pour notre exemple, nous avons deux fichiers **Makefile** corespondant chacun à une partie distincte d'un projet :

- bibliothèque de fonctions,
- programme utilisateur.

Il est utile de disposer d'un **Makefile** unique (un **Makefile** "maître") permettant d'appeler les différents **Makefile** répartis dans l'application, et, de plus, que les options soient identiques pour tous ces **Makefiles.**

La variable **MAKE** permet d'appeler un **Makefile** depuis un autre **Makefile** et de lui fournir des variables grâce à l'instruction **export.**

Si nous supposons avoir la répartition des fichiers suivante :



alors les fichiers **Makefiles** sont les suivants :

Makefile1 - Makefile "maître"

```
export SHELL = /bin/sh
export CC = gcc
export CFLAGS = -g -ansi -Wall
export OUTPUT_OPTION = -o $@

all : cible1 cible2

cible1 :
	@(cd Bib/Sources && ${MAKE})

cible2 :
	@(cd Utilisateur && ${MAKE})

clean :
	@(cd Bib && ${MAKE} $@)
	@(cd Utilisateur && ${MAKE} $@)

mpproper : clean
	@(cd Bib && ${MAKE} $@)
	@(cd Utilisateur && ${MAKE} $@)
```

Makefile2 - reconstruction de la bibliothèque

```
BIB = ../

all : ${BIB}bib.a

${BIB}bib.a : f.o g.o
	ac cur $@ $?

clean :
	-rm -f *.o

mrproper : clean
	-rm -f ${BIB}bib.a
```

Makefile2 - pour l'application

```
BIB = ../Bib
SBIB = ${BIB}/Sources
REP = /users/assemb

all : appli

appli : main.o arbre.o liste.o ${REP}es.o
	${CC} ${CFLAGS} main.o arbre.o liste.o ${REP}es.o -o $@

arbre.o : const.h glob.h

clean :
	-rm -f *.o

mrproper : clean
	-rm -f appli
```


6. Conclusion

Outil de développement : (gnu)make
└Conclusion

Limitations de make

make est un outil très puissant pour exprimer toute la connaissance d'une application sur un système, mais :

- convivialité moyenne
- syntaxe ubuesque
- pas de contrôle de flot
  les itérations doivent être effectuées au niveau du shell
- difficulté d'effectuer des modification globales
  modifier chaque Makefile, le jour où un nom de répertoire change...
- pas d'écriture facile de règles récursives

50 / 57

Outil de développement : (gnu)make
└Conclusion

ultrix 3.1 :

```
% make
cc -O -c fich.c
rm -f fich
cc -o fich fich.o -O
```

SunOs 4.1.1 :

```
% make
cc -O -pipe -target sun4 -c fich.c
rm -f fich
cc -o fich fich.o -O -pipe
```

Mips RISC/os 4.01 :

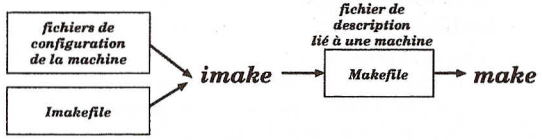
```
% make
cc -O -signed -systype bsd43 -Olimit 2000 -Wf,-XNd8400,_XNp12000 \
   -DMips -DBSD43 -c fich.c
rm -f fich
cc -o fich fich.o -O -signed -systype bsd43 -Olimit 2000 \
   -Wf,-XNd8400,_XNp12000 -lmld
```

- il est très difficile de transposer rapidement un fichier Makefile sur un autre système
- par définition, les fichiers de dépendances Makefile ne sont pas portables

51 / 57

imake

imake est un générateur de fichiers Makefile dont l'objectif est de faciliter le développement de logiciels portables sur différents systèmes.



Au lieu d'écrire un fichier ***Makefile***, il faut écrire un fichier **iMakefile** qui contient les descriptions des cibles à constuire, indépendamment des machines.

De cette façon le fichier de description ***iMakefile*** n'est plus lié au système d'exploitation utilisé.

imake combine les informations du fichier ***iMakefile*** avec les caractéristiques du système issues des fichiers de configuration et produit un fichier ***Makefile*** "sur mesure".

Le fichier ***iMakefile*** et les fichiers de configuration sont traités par le préprocesseur `cpp` ; ils en respectent la syntaxe (directives `#define`, `#ifdef`, ...).

> *"imake isn't a magic bullet that instantly and effortlessly solves all your porting woes. But it can reduce project development and maintenance tasks considerably."*

Exemple –très– simple

Soit le programme C suivant :

```
#include <stdio.h>
#include "const.h"
void main(void)
   {
     printf("bonjour !\n") ;
   }
```

Fichier Makefile écrit à la main :

```
prog : prog.o
       gcc prog.o -o prog

prog.o :  prog.c const.h
       gcc -c prog.c
```

qui fait l'exécution des deux commandes :

- `gcc -c prog.c` *règle pour construire la cible prog.o*
- `gcc prog.o -o prog` *règle pour construire la cible prog*

Fichier iMakefile

cible *liste des dépendances*

```
NormalProgramTarget(prog,   prog.o   ,NullParameter,NullParameter,\
                    NullParameter)
NormalProgramTarget(prog.o,  prog.c  ,  const.h  ,NullParameter,NullPara
```

cible *dépendance* *dépendance*

Autant de lignes `NormalProgramTarget` que de cibles et de règles de dépendances.
Création du fichier Makefile par la commande :

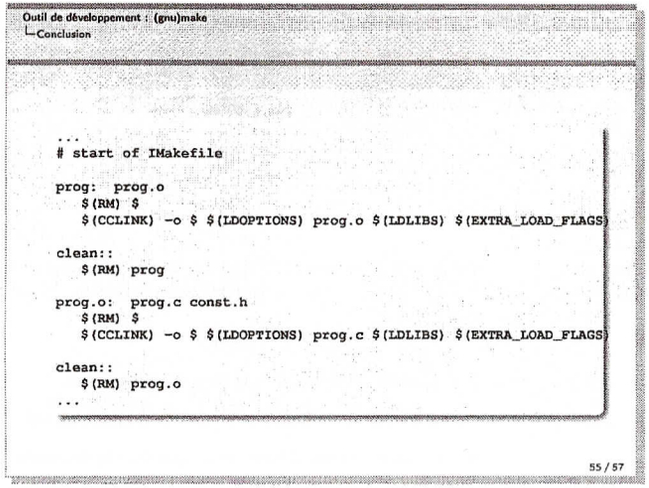
```
bash-3.00$ imake -DUseInstalled -I/usr/local/X11R6/lib/X11/config
```

où l'option `-I` précise le répertoire où les fichiers de configuration sont rangés.

Extrait de manuel en ligne de imake

```
-Ddefine
        This option is passed directly to cpp.  It is typi-
        cally used to set directory-specific variables.  For
        example, the X Window System uses this flag  to  set
        TOPDIR  to  the name of the directory containing the
        top of the core distribution and CURDIR to the  name
        of the current directory, relative to the top.
```

-Idirectory
This option is passed directly to cpp. It is typically used to indicate the directory in which the imake template and configuration files may be found.

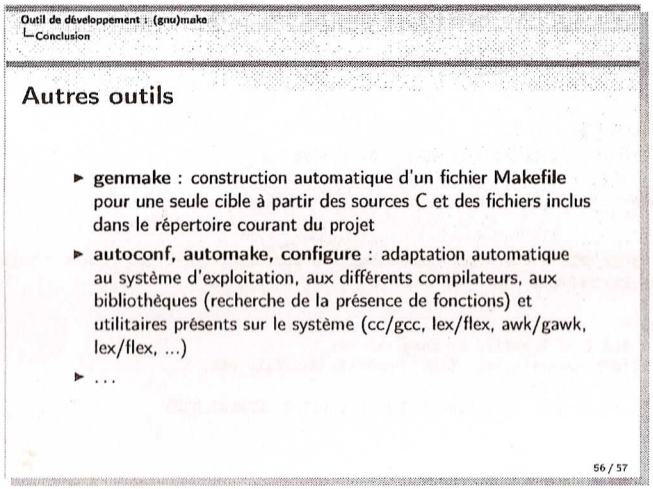


Extrait du fichier Makefile[2] construit

```
# Makefile generated by imake - do not edit!
# $XConsortium: imake.c,v 1.91 95/01/12 16:15:47 kaleb Exp $
# ----------------------------------------------------------------------
# Makefile generated from "Imake.tmpl" and <iMakefile>
# $XConsortium: Imake.tmpl,v 1.224.1.1 95/06/19 17:51:01 gildea Exp $
# $XFree86: xc/config/cf/Imake.tmpl,v 3.18 1995/07/12 15:27:23 dawes Exp $
# $Scinfo: xc/config/cf/Imake.tmpl,v 1.4 1996/11/11 17:14:16 faedi Exp $

.SUFFIXES: .i
......... <------ coupure du texte
# ----------------------------------------------------------------------
# start of IMakefile

prog: prog.o
        $(RM) $@
        $(CCLINK) -o $@ $(LDOPTIONS) prog.o  $(LDLIBS)  $(EXTRA_LOAD_FLAGS)

clean::
        $(RM) prog

prog.o: prog.c const.h
        $(RM) $@
        $(CCLINK) -o $@ $(LDOPTIONS) prog.c  $(LDLIBS)  $(EXTRA_LOAD_FLAGS)

clean::
```

[2] Makefile, avec une majuscule...


```
        $(RM) prog.o
# ----------------------------------------------------------------------
# common rules for all Makefiles - do not edit

.c.i:
        $(RM) $@
        $(CC) -E $(CFLAGS) $(_NOOP_) $*.c > $@

emptyrule::

clean::
        $(RM_CMD) *.CKP *.ln *.BAK *.bak *.o core errs ,* *~ *.a .emacs_* tags T
AGS make.log MakeOut  "#"*

Makefile::
        -@if [ -f Makefile ]; then set -x; \
        $(RM) Makefile.bak; $(MV) Makefile Makefile.bak; \
        else exit 0; fi
        $(IMAKE_CMD) -DTOPDIR=$(TOP) -DCURDIR=$(CURRENT_DIR)

tags::
        $(TAGS) -w *.[ch]
        $(TAGS) -xw *.[ch] > TAGS
# ----------------------------------------------------------------------
# empty rules for directories that do not have SUBDIRS - do not edit

install::
        @echo "install in $(CURRENT_DIR) done"

install.man::
        @echo "install.man in $(CURRENT_DIR) done"

install.linkkit::
        @echo "install.linkkit in $(CURRENT_DIR) done"

Makefiles::

includes::

depend::
# ----------------------------------------------------------------------
# dependencies generated by makedepend
```



Syntaxe de la commande make

make [**-f** *fichierDeDescription*] [*options*] [*cibles*] [*nomDeMacro=valeur*]

-b	accepte des fichiers de description d'implantation précédentes de ***make*** (compatibilité antérieure) ; cette option est prise par défaut ;
-d	mode *debug* ; affiche des informations détaillées sur les fichiers et leur date de dernière modification ;
-e	permet aux variables d'environnement d'être utilisées prioritairement sur les macros définies dans le fichier de description ;
-f *fichier*	demande à considérer *fichier* comme un fichier de description ; ceci est utile quand le fichier de description n'a pas le nom standard ***makefile*** ou ***Makefile*** ; par convention on donne le suffixe ***.mk*** aux fichiers de description portant un nom non standard ; on peut utiliser plusieurs fichiers de description, dans ce cas chacun devra être précédé de l'option ***-f*** ; ces fichiers seront concaténés ;
-f -	l'entrée standard est prise comme fichier de description ;
-i	ignore les codes d'erreur ; identique à la cible prédéfinie `.IGNORE` utilisée dans un fichier de description ;
-k	abandonne le travail sur la cible courante quand la commande en cours d'exécution retourne un code d'erreur (variable *status* différente de 0), mais continue le travail sur les autres cibles ; cette option est l'inverse de l'option `-S` ; quand les deux options `-k` et `-S` sont demandées simultanément, seule la dernière indiquée est prise en compte ;

-n	affiche seulement les commandes sans les exécuter, y compris les lignes commençant par le caractère `@` ;
-p	affiche les définitions des macros, des suffixes, des règles de suffixe ;
-q	retourne, dans la variable *status*, la valeur 0 ou une valeur non nulle selon que la cible est à jour ou non ; les cibles ne sont pas mises à jour par cette option ;
-r	efface la liste des suffixes et demande à ne pas utiliser les règles par défaut ;
-s	n'affiche pas les commandes exécutées ; identique à la cible prédéfinie `.SILENT` dans le fichier de description ;
-t	actualise la date des fichiers cibles (les faisant paraître mis à jour), au lieu d'exécuter les commandes de mises à jour ; cette option est utile quand on ajoute un commentaire dans un fichier et qu'il est inutile de recompiler.

Variables automatiques

Pour faciliter un certain nombre d'opération dans les règles, il existe des variables locales, ces variables sont parfois appelées variables automatiques Voici la liste des plus courantes:

- ***$@*** : représente le nom de la cible courante de la règle.
- ***$<*** : représente le nom du premier des fichiers de la liste de dépendance.
- ***$^*** : représente les noms des fichiers de la liste des dépendances (pour gmake).
- ***$>*** : représente les noms de l'ensemble des fichiers de la liste des dépendances (pour pmake).
- ***$?*** : représente les noms de l'ensemble des fichiers de la liste de dépendance plus récents que la cible.
- ***$**** représente le nom de la cible sans suffixe.

Cibles prédéfinies

.DEFAULT	les commandes associées à cette cible seront exécutées si ***make*** ne trouve aucune règle de dépendance ou de suffixe à l'aide de laquelle construire la cible demandée ; exemple :
	`.DEFAULT`
	`echo pas de cible à construire`

.IGNORE	ignore les codes d'erreurs, identique à l'option ***-i*** ;
.PRECIOUS	les fichiers précisés dans la définition de cette macro ne seront pas détruits même si ***make*** reçoit un signal qui conduit à sa destruction ou quand une commande d'une règle de dépendance échoue ; exemple : `.PRECIOUS: main.o argent.x`
.SILENT	exécute les commandes mais ne les affiche pas, identique à l'option ***-s*** ;
.SUFFIXES	permet de définir la liste des suffixes auxquels sont associées des règles de suffixe ; exemple : `.SUFFIXES: .o .c .s` où l'on trouve ensuite dans le fichier, les règles permettant de passer d'un suffixe à un autre.