Avant propos

Cours = Introduction aux Types de Données Abstraits (TDA)

⇒ Chaque TDA est structuré en un ensemble de fichiers

⇒ Compilation séparée pour faire les programmes

⇒ Outils :

- `make` en C
- `ant` en java ...

Dans un premier temps : utilisation du C donc de `make`

INF601 : Algorithme et Structure de données

Cous 1 : Utilitaire `make` / Compilation séparée

B. Jacob

IC2/LIUM

2 février 2010

Plan

Plan

Compilation élémentaire

Compilateur

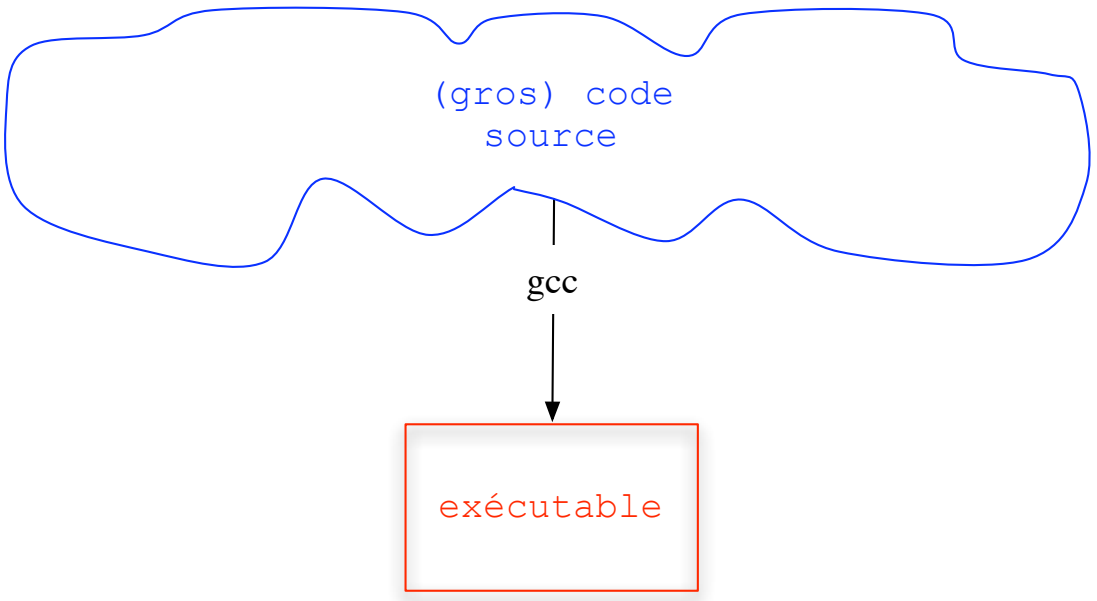
- Généralement compilateur C = `cc` ou `gcc`
- Convention : `CC` = nom du compilateur choisi

Invocation compilation

```
CC source_projet.c -o executable
```

Tout le code source d'un projet est regroupé dans un seul fichier

Compilation élémentaire



Inconvénients

- gros fichier ⇒ compilation longue lors des mises à jour
- peu lisible
- difficile à debugger et à maintenir
- réutilisation de portion de code limité

Programmation modulaire

Principe

Répartir dans plusieurs fichiers sources tout le code source d'un projet.

Intérêts

- réutilisation de certains fichiers dans d'autres projets
- projet plus structuré $\longrightarrow$ plus lisible, plus facile à maintenir

Fichiers sources

3 types de fichiers source :

Les modules :

- extension `.c`
- uniquement des fonctions (ce seront les TDA), pas de `main()`

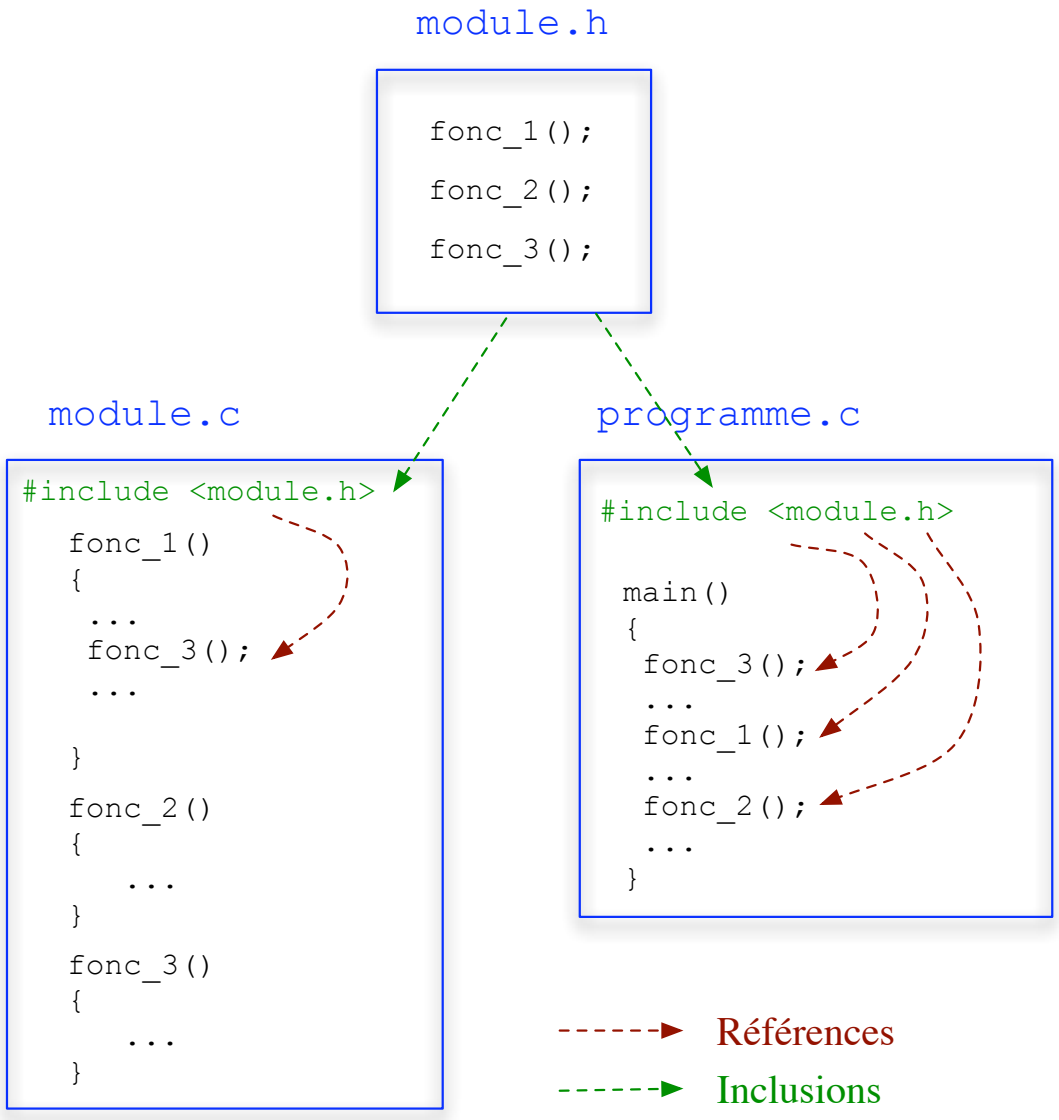
Les fichiers d'entête : ou *header*

- extension `.h`
- inclus dans les `.c` avec `#include`
- contiennent déclarations de types, les prototypes des fonctions

Les programmes :

- extension `.c`
- contiennent le `main()` et appelle les fonctions des modules

Exemple de fichiers sources



Fichiers compilables

2 type de fichiers compilables :

- les objets
- les exécutables

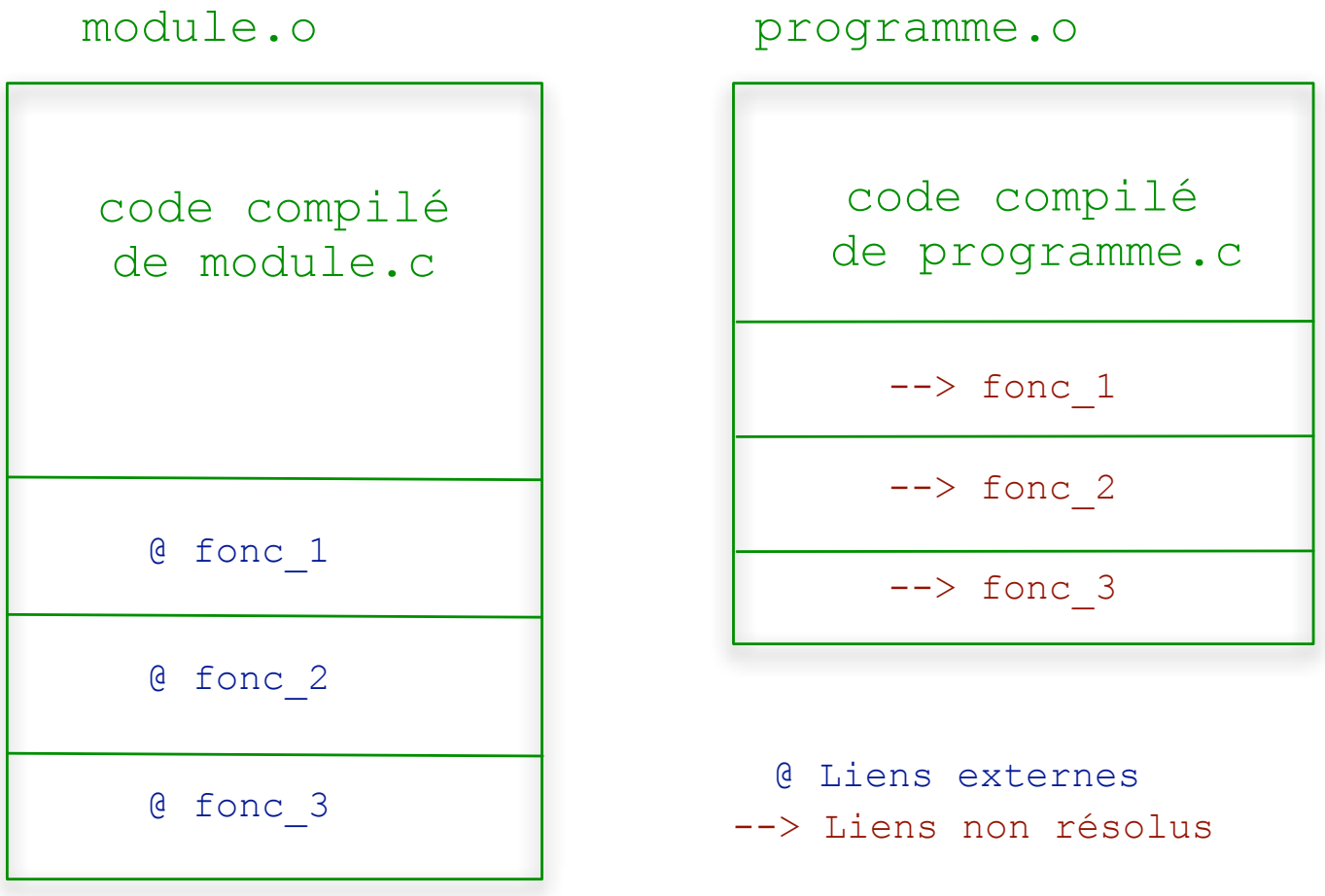
Fichiers objets

- extension `.o`
- contient le code compilé du `.c` correspondant
- table des "liens", variables/fonctions
  - exportées (définies mais pas utilisées dans le `.c`)
  - importées (pas définies mais appelées dans le `.c`)

Commande

```
CC -c mod1.c [ -o mod1.o ]
```

Exemple fichiers objets



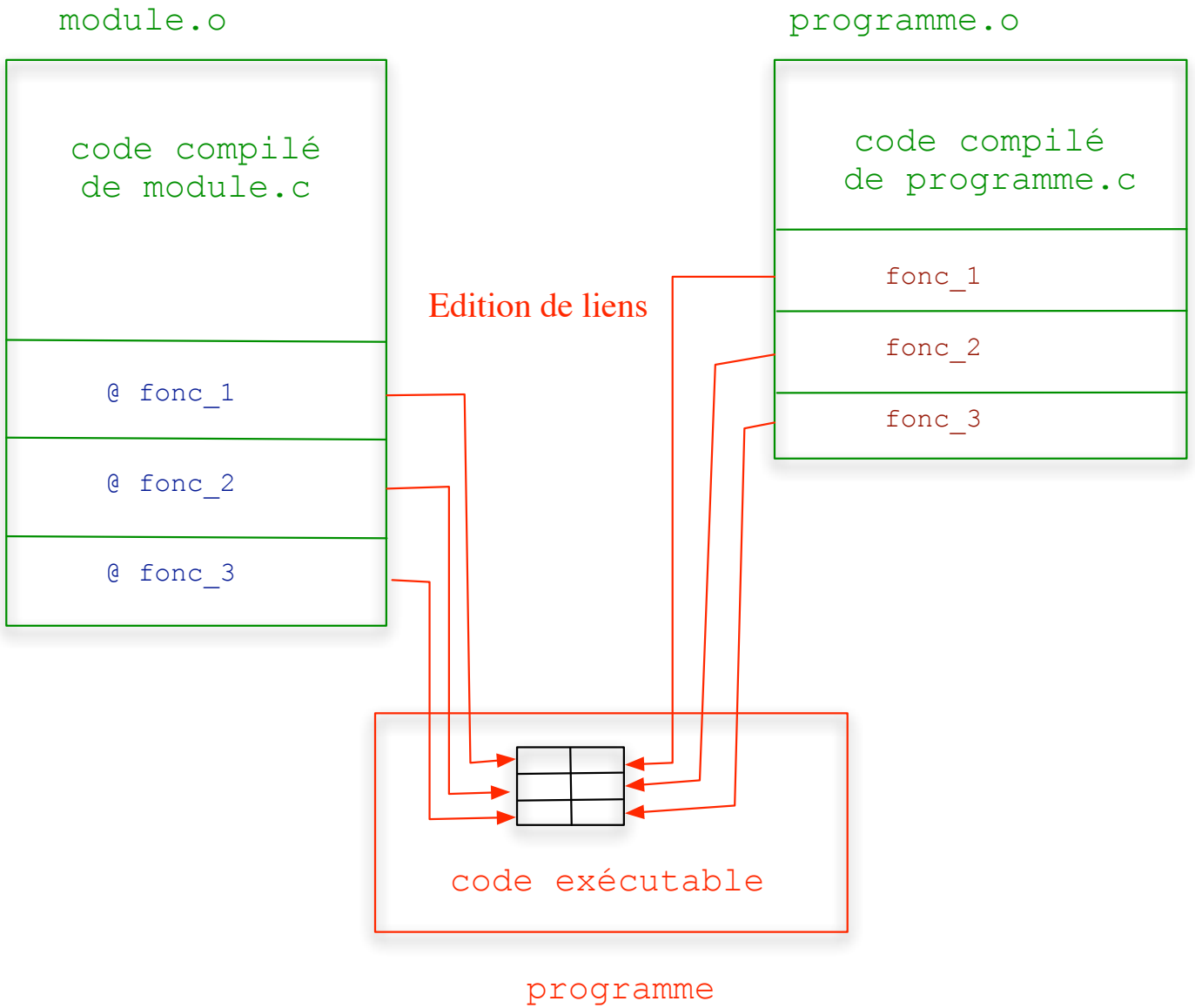
Les fichiers exécutables

- pas d'extension
- issu de l'édition de liens entre tous les objets des modules utilisés

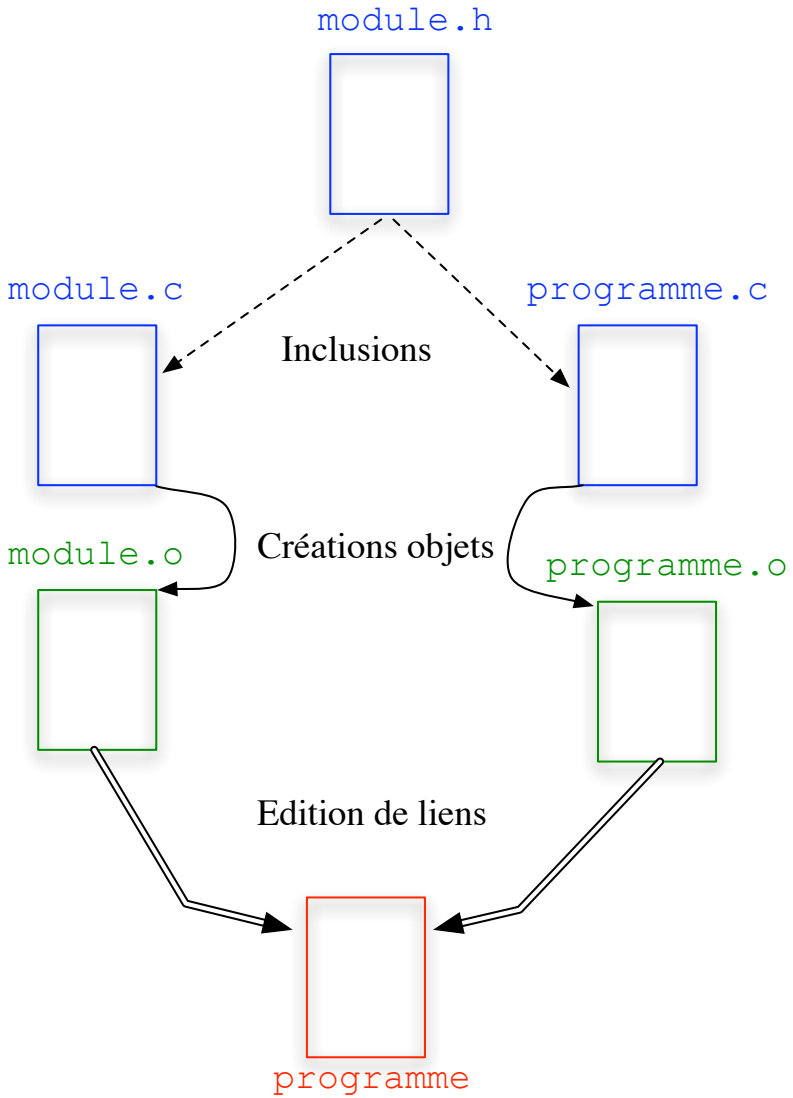
Commande

```
CC mod1.o mod2.o...modn.o -o executable
```

Exemple Création d'un exécutable



Organisation générale des fichiers



Options fréquentes de compilation

-l< *bibliotheque* > : liens avec bibliothèques (-lm)

-g : pour debuggage

-ansi : force la conformité à la norme ANSI.

-Wall : niveau des Warnings

-D< *var* = *val* > : Définition de variables du préprocesseur.

`-DDEBUG=2` ⇔ `#define DEBUG 2`

-O< *niveau* > : niveau d'Optimisation de la compilation

-I< *rep* > : chemin répertoire(s) des headers (include).

Plan

make : à quoi ça sert ?

La commande make → gestion d'un projet contenant plusieurs fichiers

- plusieurs programmes (fichiers .c avec main())
- modules (fichiers .c sans main())
- objets (exécutables translatables .o)
- bibliothèques (statiques .ar, dynamiques .so)
- des fichiers de documentation ...

De quelle manière ?

Automatisations de :

- la compilation
- création de la documentation (pdf, html, . . .)
- vos tâches fastidieuses. . .

Intérêts :

⇒ ne compile que ce qui est nécessaire

⇒ affranchit des commandes fastidieuses

⇒ facilite la distribution d'un projet . . .

Suite du cours

Objectif

Faire des projets structurés

Projet structuré
→ plusieurs fichiers
→ compilation séparée

Contenu du cour

1. commande `make` + fichier *Makefile*
2. l'utilisation de `make` dans la compilation séparée.

Principe

`make` est un outil permettant de gérer

- la mise à jour
- l'installation . . .

d'un ensemble de fichiers indépendants

Comment ? En testant leur date de dernière modification.

Principe

Son utilisation suppose que l'on ait fait

1. un fichier nommé *Makefile* ou *makefile* dans lequel on décrit
   - les dépendances entre les fichiers
   - les règles de mise à jour / création des fichiers
2. appeler la commande `make` après toute modification de l'un des fichiers sources : alors `make`
   - examine les dépendances
   - trouve les fichiers qui ne sont pas à jour
   - exécute seulement les commandes nécessaires pour les reconstruire

Principe

Un appel de la commande `make` à, généralement le format suivant

- `make -f Mon_Makefile` cible ou
- `make` cible (prend un fichier appelé `Makefile` par défaut)

cela signifie que l'on veut mettre à jour cible.

Celle cible peut être :

- le nom d'un fichier
- un label qui désigne qu'il mettre à jour un ensemble de fichiers

Invocation de make

Appels de make les plus fréquents :

- `make -f Mon_Makefile cible`
  - Mise à jour de `cible`
  - avec le fichier de règles `Mon_Makefile`
- `make` cible $\longrightarrow$ pas de règles spécifiées
  - Mise à jour de `cible`
  - avec un fichier de règles appelé `Makefile`.
    - utilise `Makefile` si $\exists$
    - utilise un `Makefile` par défaut sinon
- `make` $\longrightarrow$ pas de cible spécifiée
  - Mise à jour de toutes les cibles
  - dans l'ordre où elles apparaissent dans `Makefile`

Plan

Contenu du `Makefile`

Un *Makefile* contient

1. les règles pour mettre à jour les fichiers.
2. éventuellement des macros
3. des commentaires

Composition d'une règle

Une règle est composée de :

une 1^{iere} ligne contenant

- une cible : ce qu'il faut construire
- ses dépendances : ce dont dépend la cible

éventuellement d'autres lignes :

- instructions à faire pour reconstruire la cible

Syntaxe 1^{iere} ligne

Elle contient

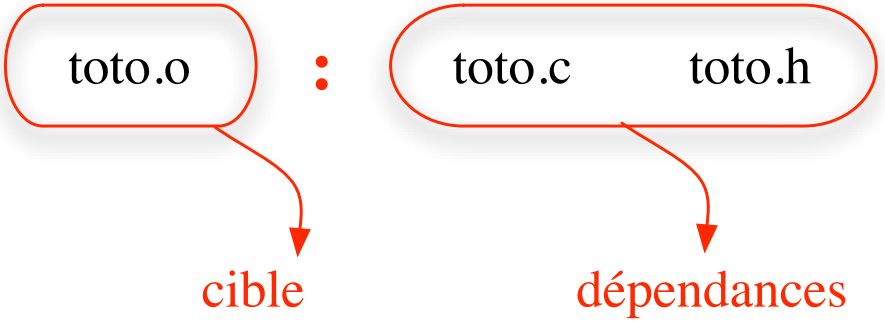
1. une cible : c'est le nom du fichier à mettre à jour ou un label
2. le caractère **:** ou **::**
3. les dépendances : une liste de fichiers dont dépend la cible

SI date dernière modification dépendance(s) $>$ date cible **ALORS**

$\Rightarrow$ il faut mettre à jour la cible.

FSI

Exemple :



Dépendances de la 1^{iere} ligne

Les dépendances sont soit

- des fichiers sources existants, "fabriqués" par l'utilisateur
  Ex : `cible : toto.c toto.h`
- des fichiers à construire (ils doivent avoir eux-mêmes une règle indiquant comment on les crée)
  Ex : `cible : toto.o module_exotique.o`
- vide : la cible ne dépend de rien et devra toujours être mise à jour
  Ex : `cible :`

Syntaxe lignes suivantes

N lignes de commandes pour reconstruire la cible

Si $N > 0$: chaque ligne doit contenir

1. le caractère de *tabulation*
2. une ou plusieurs commande(s) séparées par des **;**

Ex : $<$ *tab* $>$ `CC -c toto.c -o toto.o`

Si $N = 0$: on ne fait rien $\longrightarrow$ on vérifie juste si la cible est à jour par rapport aux dépendances

Exemples de règles

Règle simple avec des fichiers sources

```
toto.o : toto.c toto.h
	CC -c toto.c -o toto.o
```

Activation

```
make toto.o
```

SI (date `toto.c` > date `toto.o`) **OU**
(date `toto.h` > date `toto.o`) **ALORS**
on reconstruit `toto.o` par `CC -c toto.c -o toto.o`
FSI

Exemples de règles

Règle avec des fichiers qu'il faut reconstruire

```
toto : toto.o
	CC toto.o -o toto
```

Activation

```
make toto
```

1. vérification si `toto.o` est à jour avec la règle précédente
2. **SI** (date `toto.o` > date `toto`) **ALORS**
   reconstruction `toto` par `CC toto.o -o toto`
   FSI

Exemples de règles

Règle sans dépendance

```
clean :
	rm *.o
```

Activation

```
make clean
```

- la cible `clean` est toujours exécutée
- rem : `clean` souvent utilisée pour *remettre à zéro* les dépendances entre les fichiers

Exemples de règles

Règle sans ligne de commande

```
toto.o : toto.c toto.h
	CC -c toto.c -o toto.o
toto : toto.o
	CC toto.o -o toto
tata.o : tata.c tata.h
	CC -c tata.c -o tata.o
tata : tata.o
	CC tata.o -o tata
all : toto tata
```

Activation

```
make all
```

Exemples de règles

```
all : toto tata
```

⇒ vérification cible `toto`
↪ vérification cible `toto.o`
↪ vérification fichiers `toto.c` et `toto.h`
⇒ vérification cible `tata`
↪ vérification cible `tata.o`
↪ vérification fichiers `tata.c` et `tata.h`

- on en revient à vérifier toutes les dépendances
- la cible `all` est la racine de l'arbre des dépendances
- souvent utilisée pour compiler tout un projet

Intoduction aux macros

Un *Makefile* peut contenir également des macros (variables) qui peuvent être :

- définies par l'utilisateur
- prédéfinies

Définies par l'utilisateur

- Définition : `nom = valeur`
- Utilisation : `$(nom)` dans le Makefile

Exemple 1

Makefile

```
dir = /home/astre/

testAstre :
        cd $(dir)
        ls
```

Commande

```
make -f Makefile testAstre
```

Exécution

```
cd /home/astre
ls
```

Exemple 2

Makefile

```
CC = gcc
CCLNK = $(CC)
CCOBJ = $(CC) -c
fich_obj_prog1 = mod1.o mod2.o mod3.o

mod1.o : mod1.c mod1.h
         $(CCOBJ) mod1.c -o mod1.o
mod2.o : ...
mod3.o : ...

prog1 : $(fich_obj_prog1)
        $(CCLNK) $(fich_obj_prog1) -o prog1
```

Exemple 2 (suite)

Commande

```
make -f Makefile prog1
```

Exécution

```
gcc -c mod1.c -o mod1.o
gcc -c mod2.c -o mod2.o
gcc -c mod3.c -o mod3.o
gcc mod1.o mod2.o mod3.o -o prog1
```

Macros améliorent

- la lisibilité
- la saisie des règles

Noms de macros prédéfinies

- Utilisables dans la 2^{ieme} ligne des règles (ou de production)
- Les plus courantes :
  - `$@` = nom de la cible
  - `$*` = nom de la cible sans son extension
  - `$?` = fichiers des dépendances plus récents que la cible

Exemple utilisation macros prédéfinies

```
prog1 : $(fich_obj_prog1)
        echo "fichiers plus recents que " $@ " = " $?

prog1.o : prog1.c
        echo "racine cible = " $*
```

Insertion de commentaires

Tout ce qui est après un #

Exemple

```
# ceci un un commentaire
V1 = coucou  # et ceci en est un autre
V2 = salut # commentaire # V3 = ciao commentaire aussi
```

Plan

Introduction

Dans un projet, le Makefile permet :

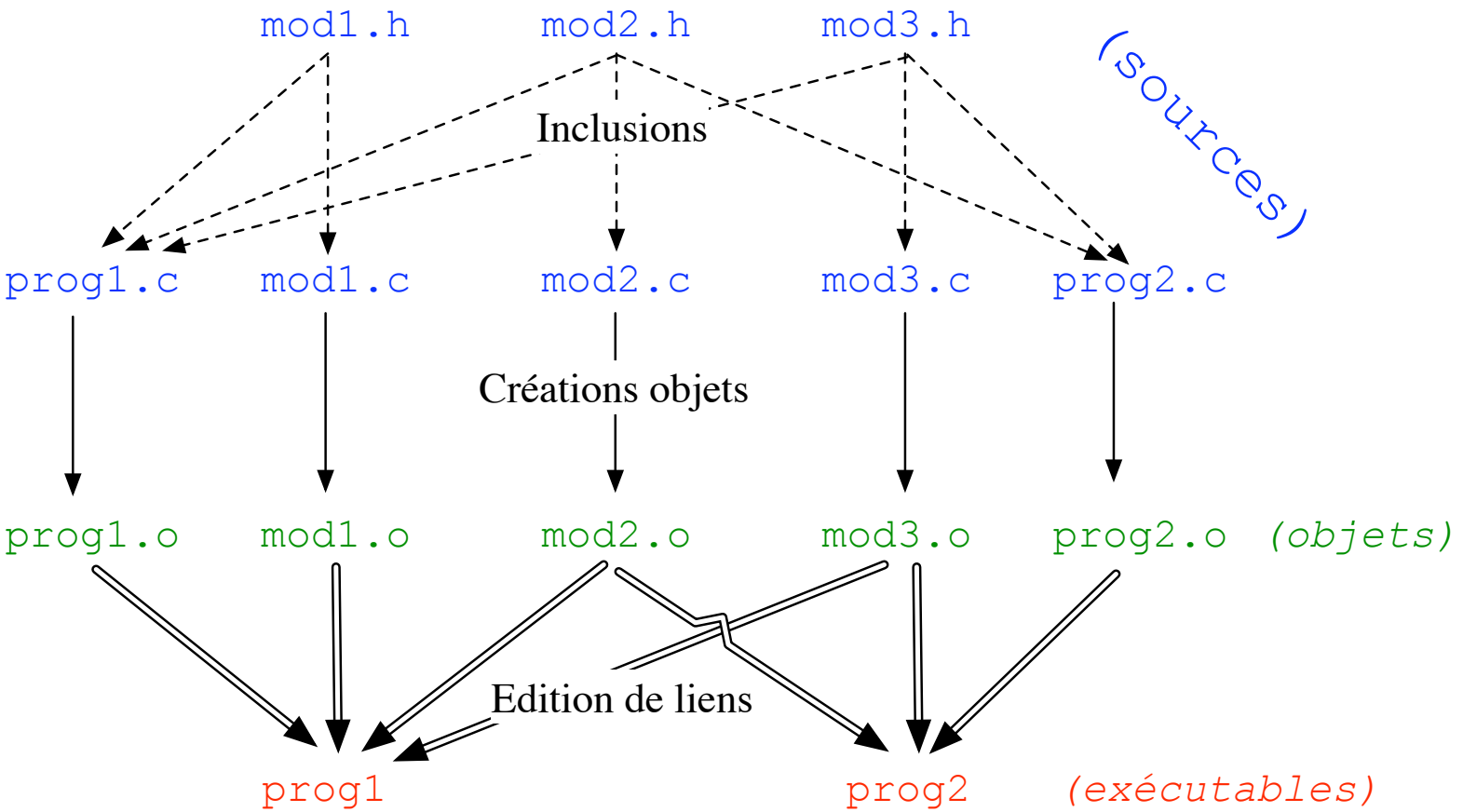
- de garder les dépendances entre les fichiers
- d'automatiser la compilation (compiler un à un les fichiers en ligne de commande serait fastidieux)
- mais dépasse largement la compilation de programmes en C

Structure du projet

Liste des fichiers du projet :

- 2 programmes : `prog1` et `prog2`
- 3 Modules (fichiers C sans `main`) : `mod1.{c,h}` , `mod2.{c,h}` et `mod3.{c,h}`
- 1 documentation `manuel.tex`
- Les règles de dépendance suivantes :
  - `prog1` utilise des fonctions définies dans `mod1` , `mod2` et `mod3`
  - `prog2` utilise des fonctions définies dans `mod2` et `mod3`
  - on veut créer une documentation en pdf et en ps. Il faut donc faire :
    - `latex manuel.tex ; dvips manuel.dvi -o manuel.ps` pour créer la doc en ps
    - `ps2pdf manuel.tex` pour créer la doc en pdf

Dépendances entre les fichiers



Les macros

```
#
#--- Macros
#
CCLNK = gcc
CCOBJ = gcc -c
PS2PDF = ps2pdf
DVI2PS = dvips
TEX2DVI = latex
fich_obj_prog1 = prog1.o mod1.o mod2.o mod3.o
fich_obj_prog2 = prog2.o mod2.o mod3.o
```

Les règles des exécutables

```
#
#--- Programmes
#
prog1 : $(fich_obj_prog1)
        $(CCLNK) $(fich_obj_prog1) -o prog1
prog1.o : prog1.c
        $(CCOBJ) prog1.c -o prog1.o

prog2 : $(fich_obj_prog2)
        $(CCLNK) $(fich_obj_prog2) -o prog2
prog2.o : prog2.c
        $(CCOBJ) prog2.c -o prog2.o
```

Les règles des modules

```
#
#--- Modules
#
mod1.o : mod1.c mod1.h
	$(CCOBJ) mod1.c -o mod1.o

mod2.o : mod2.c mod2.h
	$(CCOBJ) mod2.c -o mod2.o

mod3.o : mod3.c mod3.h
	$(CCOBJ) mod3.c -o mod3.o
```

Les règles de la documentation

```
#
#-- Documentation
#
manuel.pdf : manuel.ps
	$(PS2PDF) manuel.tex

manuel.ps : manuel.dvi
	$(DVI2PS) manuel.dvi -o manuel.ps

manuel.dvi : manuel.tex
	$(TEX2DVI) manuel.tex
```

Les règles générales

```
#
#-- Utilitaires generaux
#
all : prog1 prog2

clean :
        rm *.o
doc :
        $(TEX2DVI) manuel.tex
        $(DVI2PS) manuel.dvi -o manuel.ps
        $(PS2PDF) manuel.ps manuel.pdf
```

Règles avec macros prédéfinies

```
#
#--- Programmes
#
prog1 : $(fich_obj_prog1)
        $(CCLNK) $(fich_obj_prog1) -o $@
prog1.o : prog1.c
        $(CCOBJ) $*.c -o prog1.o
.....
#
#--- Modules
#
mod1.o : mod1.c mod1.h
        $(CCOBJ) $*.c -o $@
.....
```

Plan

Substitution de suffixes

⇒ Substitution automatique des extensions les noms de fichiers contenus dans des macros

Exemples :

1. substitution de `.old` par `.new` dans la macro `NOM`

   ```
   $(NOM:.old=.new)
   ```

2. création de la liste des objets à partir de celle des sources

   ```
   SOURCES= prog1.c prog2.c mod1.c mod2.c mod3.c
   OBJETS=$(SRCS:.c=.o)
   ```

Caractères spéciaux de début de ligne

Comportement par défaut de `make` :

- affichage de toutes les lignes de commandes avant de les exécuter
- arrêt immédiat quand il y a une erreur sur une ligne de commande
  $\longrightarrow$ n'exécute pas les lignes suivantes

Les caractères suivants, placés en début d'une ligne de commandes, modifient localement (juste pour la ligne) le comportement de `make`

- **@** : pas d'affichage de la commande avant l'exécution
- **-** : en cas d'erreur, poursuite du `make`

Exemples comportement par défaut

Makefile

```
test :
        echo "coucou"
        cd ...
        ls
```

Production de make test

```
echo "coucou"
coucou
cd ...
/bin/sh: ...: n'existe pas.
make: *** [test] Error 1
```

Exemple comportement avec caractères spéciaux

Makefile

```
test :
        @ echo "coucou"
        - cd ...
        ls
```

Production de make test

```
coucou
cd ...
/bin/sh: ...: n'existe pas.
make: [test] Error 1 (ignored)
ls
mod1.c                          mod1.h
mod2.c                          mod2.h
....
```

Généralisation du comportement

Si l'on veut avoir ces comportements par défaut dans tout le Makefile alors on peut mettre les lignes suivantes :

- `.IGNORE` : pas d'arrêt sur erreur
- `.SILENT` : pas d'affichage des commandes

Règles de suffixes

Une seule règle pour :

- créer tous les fichiers `.sfx2`
- à partir des fichiers `.sfx1`

Syntaxe

`.sfx1.sfx2 :`

fichier `.sfx1` –[commande]→ *fichier* `.sfx2`

Exemple règle de suffixes

Syntaxe

```
.c.o :
        gcc -c $< -o $@
```

Effet

Transforme les fichiers sources (`.c`) en fichiers objets (`.o`)

Avec :

- `$<` = fichier de dépendance (ou fichier en entrée)
- `$@` = fichier cible (ou fichier en sortie)

Exécution de `make toto.o`

```
gcc -c toto.c -o toto.o
```

Conditions d'utilisation des règles de suffixes

- **Attention :** pas de dépendance pour une règle de suffixe
  $\longrightarrow$ serait ignorée ou pas interprétée comme une règle de dépendance

- Les règles de suffixes les plus usuelles sont déjà définies dans `/usr/share/lib/make/make.rules`
  Ce sont les *règles universelles*
  On peut les consulter en faisant `make -p`

Makefile avec règles de suffixes

```
#-- Regle suffixe pour compilation des objets
.c.o :
        $(CCOBJ) $< -o $@

#-- Documentation
.ps.pdf :
        $(PS2PDF) $< $@
.dvi.ps :
        $(DVI2PS) $< -o $@
...
```

Perte dépendances

Mais :

- perte de la dépendance entre les `.c` et les `.h`.
- il faudrait rajouter les dépendances suivantes :

```
mod1.o: mod1.h
mod2.o: mod2.h
mod3.o: mod3.h
prog1.o: mod1.h mod2.h mod3.h
prog2.o: mod2.h mod3.h
```

⇒ commande `makedepend`

L'utilitaire `makedepend`

`makedepend`

- écrit dans un Makefile des règles de dépendance entre les `.o` et les `.h` inclus dans les `.c`
- utilise le préprocesseur C (`#include`).
- ajoute les nouvelles dépendances en bloc à la fin du Makefile

Invocation

```
makedepend *.c -f Mon_Makefile
```

Utilisation de `makedepend`

- La solution généralement adoptée pour utiliser `makedepend` est d'inclure dans les lignes du Makefile une règle du style :

```
SRC = liste des fichiers sources

depend :
        makedepend $(SRC)
```

- Faire un `man makedepend` pour plus d'information

Caractères jokers

- Certaines versions de `make` admettent des règles avec le caractère joker **%**
- mais pas standard
  (déconseillé dans les makefiles distribués)

```
%.dvi : %.tex
        latex $<
%.ps : %.dvi
        dvips $< -o $@
%.o : %.c %.h
        $(CCOBJ) $< -o $@
```

Arborescence de Makefiles

Si projet organisé en arborescence de répertoires alors
$\Rightarrow$ arborescence de Makefiles
Convention :

- un à la racine
- un dans chaque répertoire

Organisation des Makefiles

Makefile racine

Rôle :

- Définit les macros communes
- invoque les makefiles *fils*

Comment ?

- appel de `make -f Répertoire_fils/Makefile...`
  ou `cd Répertoire_fils ; make -f Makefile...`
- inclusions par `include`
- sélection par `ifeq, else, endif`

Makefiles *fils*

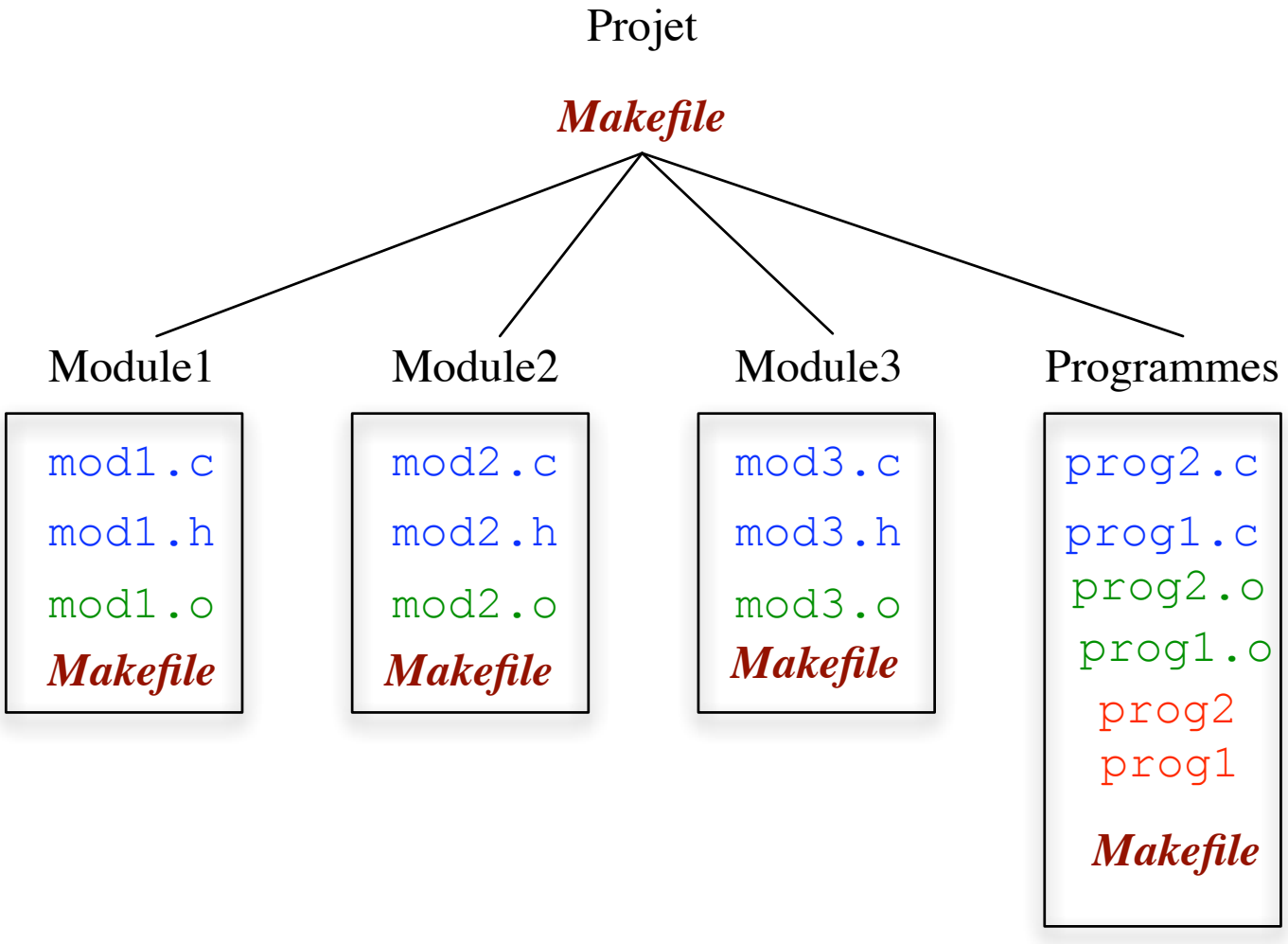
idem Makefiles précédents

Exemple : répartition des fichiers

Projet structuré en 4 Répertoires :

```
ls -R Projet

./Module1:
mod1.c   mod1.h

./Module2:
mod2.c   mod2.h

./Module3:
mod3.c   mod3.h

./Programmes:
prog1.c prog2.c
```

Exemple : arborescence des répertoires



⇝ voir exemple Projet

Automatisation de la création des Makefiles

A partir de 2 outils principaux :

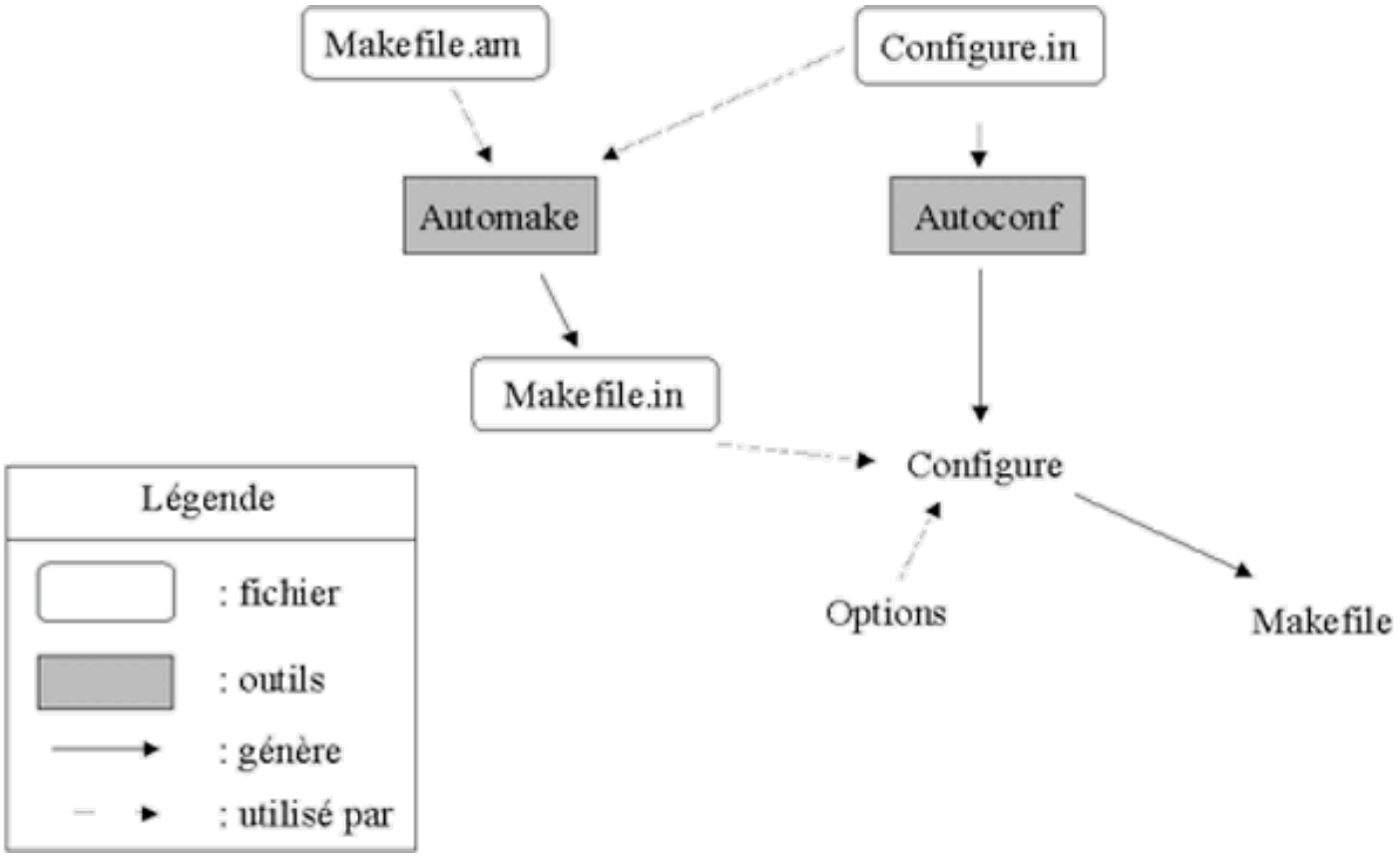
Automake : pour créer le Makefile. Basé sur :

- d'un Makefile simplifié qui décrit la structure de l'application
- de macros prédéfinies

Autoconf : pour que le projet soit portable. Basé sur des programmes qui testent :

- le type de machine
- les librairies installées...

Organisation simplifiée Automake / Autoconf



Automake / Autoconf

- Organisation réelle plus complexe
- Système élaboré empiriquement → un peu lourd à mettre en place
- Utilisation poussée des macros prédéfinies → Makefile peu lisible
- Destiné à des programmeurs avertis qui veulent distribuer leur projet
  → Automake pas demandé dans la suite des TPs mais
  → Connaissance du Makefile requise

The End

That all folks. . .