

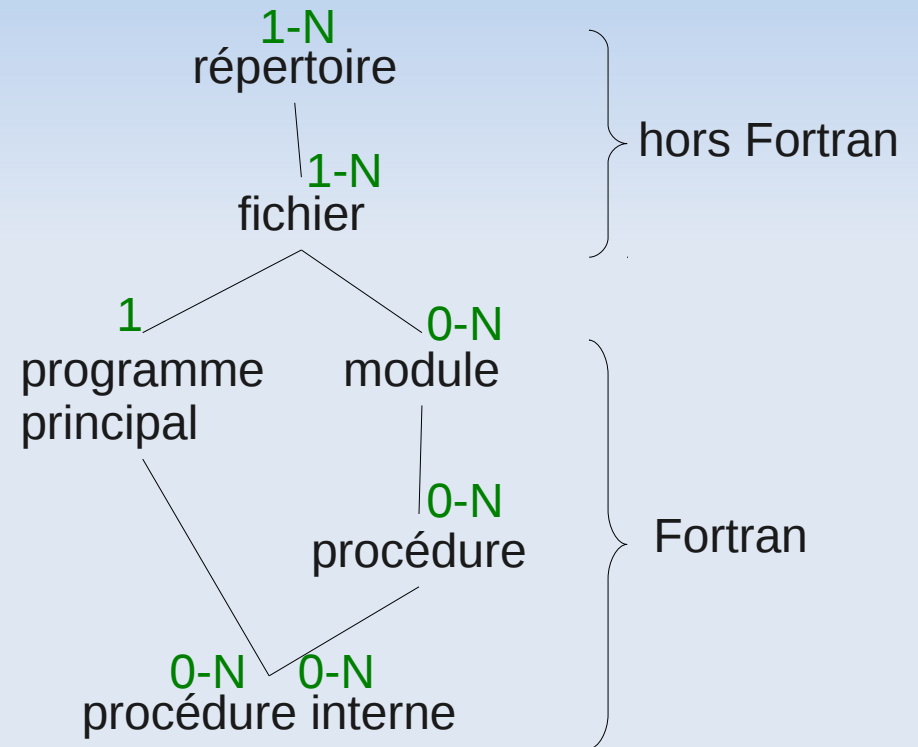
# Architecture d'un programme en Fortran, compilation, make

École normale supérieure  
L3 sciences de la planète Terre  
2011/2012  
Version du 2 février 2012

Lionel GUEZ

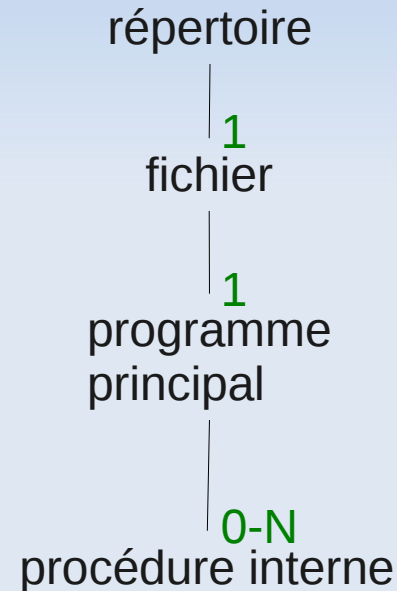
# Niveaux de structuration d'un programme en Fortran

- Structuration selon :  
clarté, confort,  
cacher ou non des  
procédures,  
contraintes du  
langage...
- Problème de  
"l'architecture du  
programme"



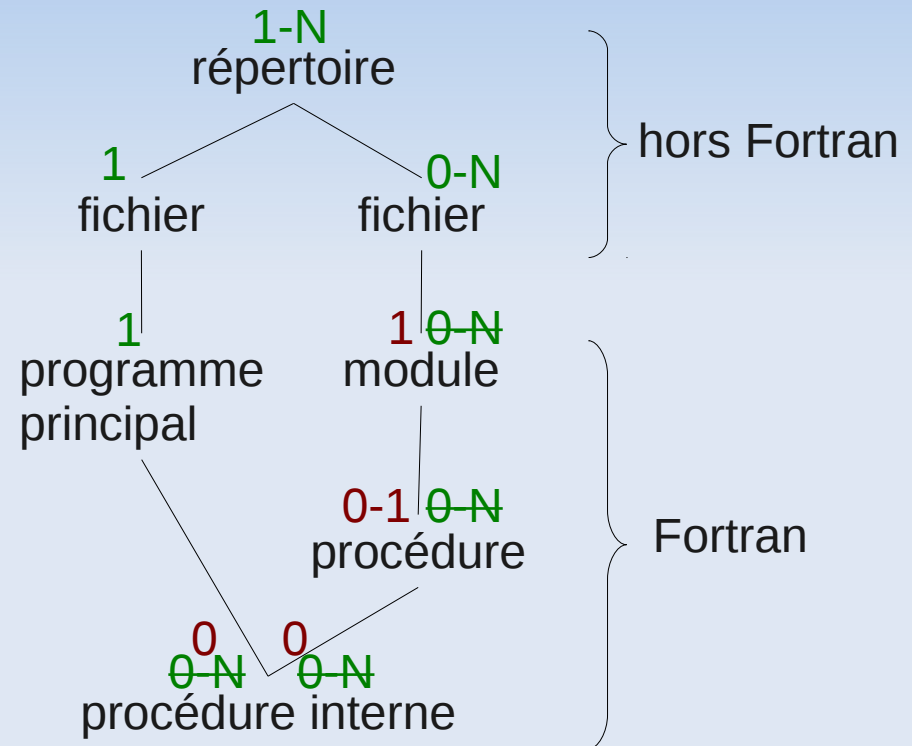
# Jusqu'à maintenant

- Un seul fichier à compiler



# Conseil de structuration d'un programme en Fortran

- Le plus clair, sauf contrainte spéciale :  
1 fichier pour programme principal  
+ (1 fichier, 1 module, 0-1 procédure) \* (0-N)



# Noms de fichiers etc.

- Des noms à choisir pour : fichiers, programme principal, modules, procédures
- 1 fichier pour programme principal + (1 fichier, 1 module, 0-1 procédure) \* (0-N) → choisir une convention simple
- Nom de fichier pour programme en Fortran, convention : suffixe **.f**

# Noms de fichiers etc. : conseil

my\_program.f

```
program my_program  
...
```

foo.f

```
module foo_m  
...  
contains  
[subroutine | function] foo  
...
```

bar.f

```
module bar_m  
...  
contains  
[subroutine | function] bar  
...
```

- Contrainte : nom de module  $\neq$  nom de procédure

# Déclaration use

my\_program.f

```
program my_program
  use foo_m, only: foo
  ...
  call foo(...)
```

- **use** nom de module, **only:** nom de procédure

foo.f

```
module foo_m
  ...
  contains
  subroutine foo(...)
  use bar_m, only: bar
  ...
  ... = bar(...) + ...
```

bar.f

```
module bar_m
  ...
  contains
  ... function bar(...)
  ...
```

# Fichiers sources et exécutable

- Texte du programme en Fortran : "fichiers sources"
- Compilation : traduction du Fortran en "langage machine", langage du processeur  
→ création d'un fichier "exécutable"



# Plusieurs fichiers à compiler

- Compilation séparée de chaque fichier : une commande de compilation par fichier
- Option **-c** (tous les compilateurs) : création d'un "fichier objet", en langage machine, non exécutable, suffixe **.o**

fichier1.f — gfortan -c fichier1.f ► fichier1.o (+ ...)

fichier2.f — gfortan -c fichier2.f ► fichier2.o (+ ...)

“fichiers objets”

# Ordre de compilation

- Règle : compilation du module utilisé avant compilation du module utilisateur ou du programme principal utilisateur

plouf.f

```
program plouf
  use foo_m, only: foo
  ...
```

```
gfortran -c bar.f
gfortran -c foo.f
gfortran -c plouf.f
```

foo.f

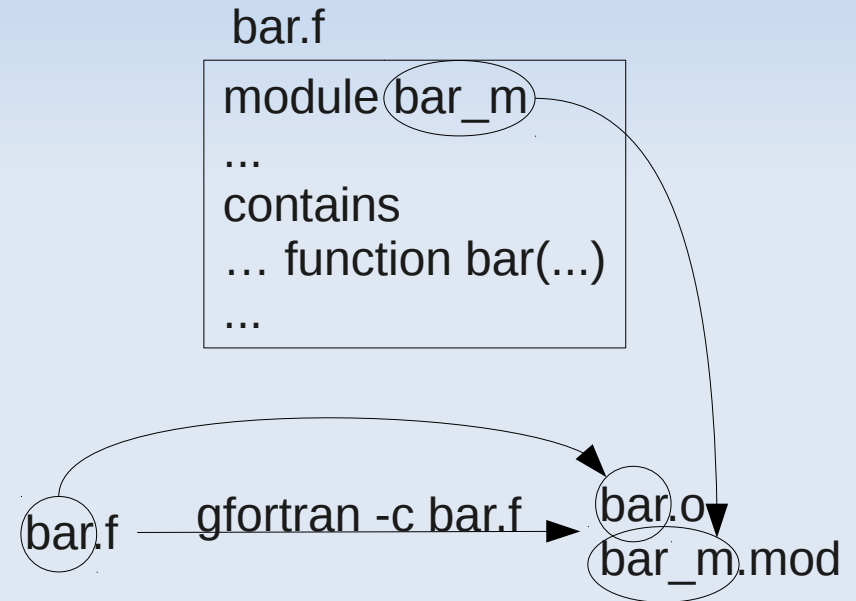
```
module foo_m
  ...
  contains
  subroutine foo(...)
    use bar_m, only: bar
    ...
```

bar.f

```
module bar_m
  ...
  contains
  ... function bar(...)
  ...
```

# Fichiers .mod

- Compilation crée un fichier "interface de module compilée" par module, suffixe **.mod** en général (peut dépendre du compilateur)



# Fichiers .mod (suite)

- Fichier `.mod` permet au compilateur de connaître l'interface de la procédure appelée
- Importance des attributs (`optional`, `pointer`...), façon dont sont déclarés les tableaux, etc. pour traduction en langage machine
- Vérification à la compilation de la validité de l'appel de la procédure (types, scalaires, tableaux...)

# Édition de liens

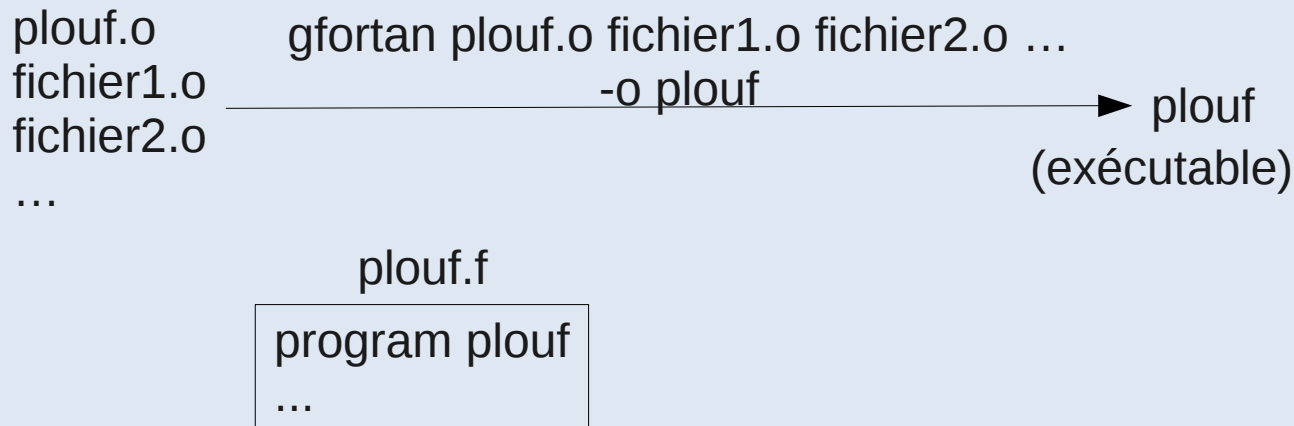
- Après la compilation, rassemblement des fichiers objets pour produire un exécutable

```
plouf.o      gfortan plouf.o fichier1.o fichier2.o ...  
fichier1.o  _____ -o plouf _____▶ plouf  
fichier2.o  
...
```

- Fichiers **.mod** non utilisés à l'édition de liens
- Exécution :  
chemin/**plouf**

# Nom de l'exécutable

- Conseil : nom de l'exécutable = nom du programme principal = nom du fichier contenant le programme principal sans **.f**



# Informations pour compiler

- Liste des fichiers sources
- Ordre de compilation
- Bibliothèques utilisées
- Nom de l'exécutable
- Choix du compilateur
- Options de compilation

# make

- Utilitaire UNIX (comme `ls`, `cd`, `awk`...)
- Vocation première : aide à la compilation
  - Stockage dans un fichier, appelé `Makefile`, de toutes les informations nécessaires à la compilation d'un programme, sous une forme directement utilisable
  - `make` déduit l'ordre général de compilation à partir de dépendances
  - `make` ne compile que le nécessaire à partir des dates de modification

# Notion de dépendance dans make

- fichier1 dépend de fichier2 :
  - fichier2 doit exister pour pouvoir créer fichier1
  - fichier1 n'est pas à jour si fichier2 modifié plus récemment que fichier1
  - Syntaxe dans le makefile :  
`fichier1: fichier2`

# Dépendance dans make pour programme en Fortran

- fichier1.o: fichier2.o

plouf.f

```
program plouf
use foo_m, only: foo
use bar_m, only: bar
...
```

foo.f

```
module foo_m
...
contains
subroutine foo(...)
use bar_m, only: bar
...
```

bar.f

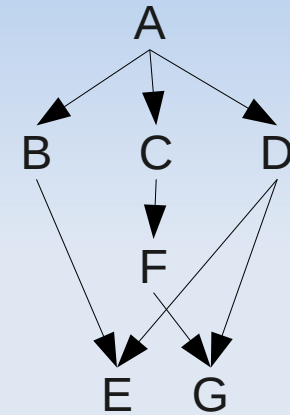
```
module bar_m
...
contains
... function bar(...)
...
```

plouf.o: foo.o bar.o  
foo.o: bar.o

# Arbre des dépendances

- Fonctionnement interne de **make** : construction de l'arbre des dépendances à partir de la liste des dépendances, déduit l'ordre de création des fichiers

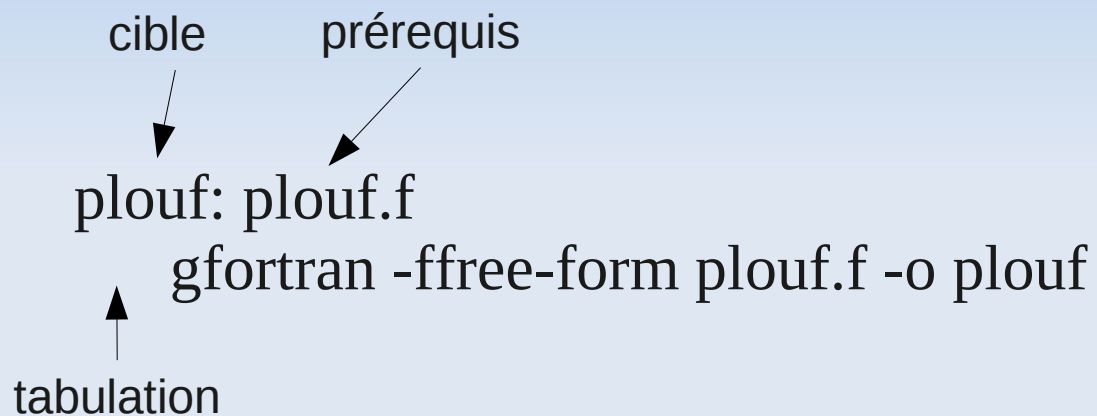
B: E  
F: G  
D: G E  
C: F  
A: B C D



Ordre : G, E, F, D, C, B, A

# Notion de règle dans make

- Définition d'une "règle" dans make : une dépendance et une commande
- La commande dit comment mettre à jour la cible



# VARIABLES RECONNUES PAR MAKE

- **FC** : *Fortran compiler*, nom du compilateur (à utiliser dans la commande de compilation)

Exemple :

```
FC = gfortran
```

- **FFLAGS** : *Fortran flags*, options de compilation

Exemple :

```
FFLAGS = -ffree-form -std=f95  
-Wall -fbounds-check
```

# Règles implicites de make

- **make** sait faire beaucoup de choses tout seul  
Règles implicites basées sur le suffixe des fichiers
- Exemple, programme à un seul fichier

Makefile

```
FC=gfortran  
FFLAGS=-ffree-form
```

plouf.f  
↑  
suffixe .f

**make** sait déjà :

```
plouf: plouf.f
```

```
    $(FC) $(FFLAGS) plouf.f -o plouf
```

donc **make plouf** (sans suffixe)

donne :

```
gfortran -ffree-form plouf.f -o plouf
```

# Règles implicites de make (suite)

- make sait déjà faire un objet `.o` à partir d'un fichier source `.f`

```
fichier1.o: fichier1.f
```

```
$(FC) $(FFLAGS) -c fichier1.f
```

# Modèle de makefile

- Remplacer les points de suspension
- **make** tout court pour créer l'exécutable
- **make clean** pour effacer ce que le makefile peut recréer

```
FC = ...
FFLAGS = ...
sources = ...
# (Write source file names separated by
blanks, not commas)

# Executable file:
execut = ...

objects = $(sources:.f=.o)

${execut}: ${objects}
    $(FC) $(LDFLAGS) $^ $(LDLIBS) -o $@

.PHONY: clean

clean:
    rm -f ${execut} ${objects}

# Dependencies between object files:
....0: ....0
```