

# Projet de Programmation

Benoit Donnet  
Année Académique 2023-2024



1

## Agenda

### Partie 2: Outils

- Chapitre 1: Compilation
- Chapitre 2: Librairie
- Chapitre 3: Tests
- Chapitre 4: Documentation
- Chapitre 5: Débogage
- Chapitre 6: Gestion des Versions

# Agenda

- Chapitre 1: Compilation
  - Compilation Multi-Fichiers
  - make
  - Arborescence

# Agenda

- Chapitre 1: Compilation
  - Compilation Multi-Fichiers
    - ✓ Compilation Élémentaire
    - ✓ Programmation Modulaire
    - ✓ Pré-Traitement
    - ✓ Fichiers Compilables
    - ✓ Vision Globale
    - ✓ Options
    - ✓ Warnings
    - ✓ Efficacité
  - Make
  - Arborescence

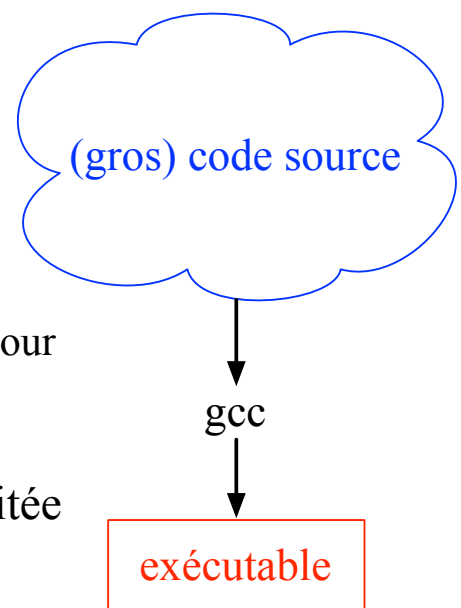
# Compilation Elém.

- Généralement, le compilateur C est `gcc` ou `cc`
- Si tout le code source d'un projet est regroupé au sein d'un seul fichier

```
$> gcc -o exécutable source_projet.c
```

## Compilation Elém. (2)

- Inconvénients?
  - gros fichier
    - ✓ compilation longue lors des mises à jour
  - peu lisible
  - difficile à débbuger et à maintenir
  - réutilisation de portion de code limitée



# Prog. Modulaire

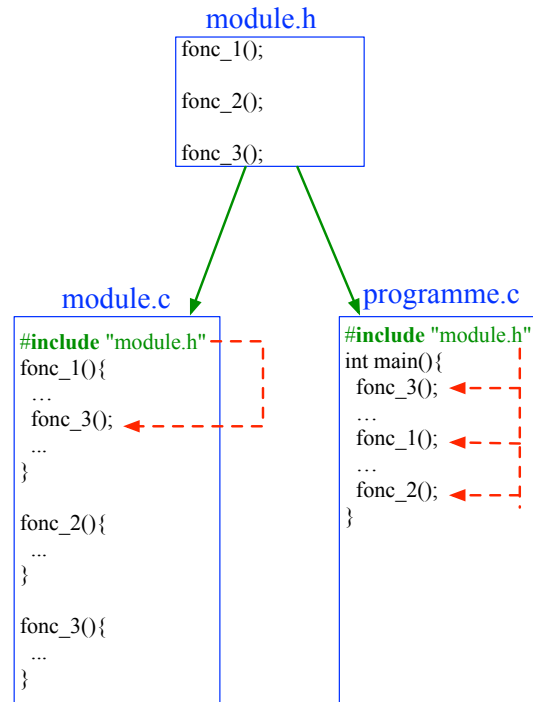
- Principe
  - répartir dans plusieurs fichiers sources tout le code source d'un projet
  - *découpe en sous-problèmes*
- Intérêts?
  - réutilisabilité
    - ✓ certains fichiers peuvent être réutilisés dans d'autres projets
  - projet plus structuré
    - ✓ meilleure lisibilité
    - ✓ maintenance plus facile

## Prog. Modulaire (2)

- Il existe trois types de fichiers source
  - **module**
    - ✓ extension `.c`
    - ✓ uniquement l'implémentation des fonctions/procédures
    - ✓ pas de `main()`
  - **fichier d'en-tête**
    - ✓ extension `.h`
    - ✓ inclus dans les `.c` avec les dérives de compilation (`#include`)
    - ✓ déclaration de type et interface des fonctions/procédures
  - **programme**
    - ✓ extension `.c`
    - ✓ contient le `main()`
    - ✓ appelle les fonctions/procédures des modules

# Prog. Modulaire (2)

- Exemple de fichiers sources



## Pré-Traitement

- La première étape de la compilation est une étape de **pré-traitement**
  - le code source est "pré-traité"
  - pré-processing**
- gcc appelle, automatiquement, l'utilitaire cpp
- Objectifs?
  - élimination des commentaires
    - ✓ inutile pour générer le code cible
  - remplacement des macros
    - ✓ substitution textuelle
  - inclusion de sous-fichiers

# Pré-Traitement (2)

- Elimination des commentaires

```
/*
 * Ceci est un commentaire décrivant mon programme.
 *
 * J'aime l'informatique, j'aime le Prof., j'aime le Rock.
 */

int main(){
    //Ceci est un commentaire
    int a = 5;

    /* Ceci est un autre commentaire */
    int b = 6;
    int c = a+b;

    return 0;
} //fin programme
```

# Pré-Traitement (3)

- Elimination des commentaires (cont.)

```
$> gcc -E precompilateur_commentaires.c
# 1 "precompilateur_commentaires.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 170 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "precompilateur_commentaires.c" 2

int main(){

    int a = 5;

    int b = 6;
    int c = a+b;

    return 0;
}
```

# Pré-Traitement (4)

- Remplacement macro

```
#define MA_CONSTANTE 8
#define PI 3.14159

int main(){
    int a = MA_CONSTANTE;
    int b=MA_CONSTANTE + 8;
    int tableau[MA_CONSTANTE];
    float rayon = 3.0;
    float aire = rayon * PI;

    return 0;
} //fin programme
```

# Pré-Traitement (5)

- Remplacement Macro (cont.)

```
$> gcc -E precompilateur_macro.c
# 1 "precompilateur_macro.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 170 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "precompilateur_macro.c" 2

int main(){
    int a = 8;
    int b=8 + 8;
    int tableau[8];
    float rayon = 3.0;
    float aire = rayon * 3.14159;

    return 0;
}
```

# Pré-Traitement (6)

- Remplacement macro (cont.)

```
#define MA_CONSTANTE 8;
#define PI 3.14159;

int main(){
    int a = MA_CONSTANTE;
    int b=MA_CONSTANTE + 8;
    int tableau[MA_CONSTANTE];
    float rayon = 3.0;
    float aire = rayon * PI;

    return 0;
} //fin programme
```

```
# 1 "precompilateur_macro.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 170 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "precompilateur_macro.c" 2

int main(){
    int a = 8;;
    int b=8; + 8;
    int tableau[8;];
    float rayon = 3.0;
    float aire = rayon * 3.14159;;

    return 0;
}
```

# Pré-Traitement (7)

- Inclusion sous-fichiers

```
int carre(int x);
```

```
#include "precompilateur_inclusion.h"
```

```
int main(){
    int x = carre(5);

    return 0;
} //fin programme
```

```
int carre(int x){
    return x*x;
} //fin carre()
```



# Pré-Traitement (8)

- Inclusion sous-fichier (cont.)

```
$> gcc -E precompilateur_inclusion.c
# 1 "precompilateur_inclusion.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 170 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "precompilateur_inclusion.c" 2
# 1 "./precompilateur_inclusion.h" 1
int carre(int x);
# 2 "precompilateur_inclusion.c" 2

int main(){
    int x = carre(5);
    return 0;
}
int carre(int x){
    return x*x;
}
```

# Pré-Traitement (9)

- Le pré-traitement est potentiellement puissant
  - va au-delà du langage C
- A utiliser avec précaution
  - rend rapidement le code peu standard
  - donc peu lisible
- Bonne pratique: n'utilisez que les règles standards!
  - include guards
    - ✓ cfr. Partie 1
  - compilation conditionnelle
    - ✓ portabilité, debug
    - ✓ cfr. Chap. 5
  - pas d'optimisation
    - ✓ préférez les fonctions/procédures aux macros

# Fichiers Compilables

- Il existe deux types de fichiers compilables
  - les objets
  - les exécutables

## Fichiers Compilables (2)

- Fichier **objet**
  - 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

```
$> gcc -c module.c [-o module.o]
```

# Fichiers Compilables (2)

module.o

code compilé  
de module.c

@fonc\_1

@fonc\_2

@fonc\_3

programme.o

code compilé  
de programme.c

↪fonc\_1

↪fonc\_2

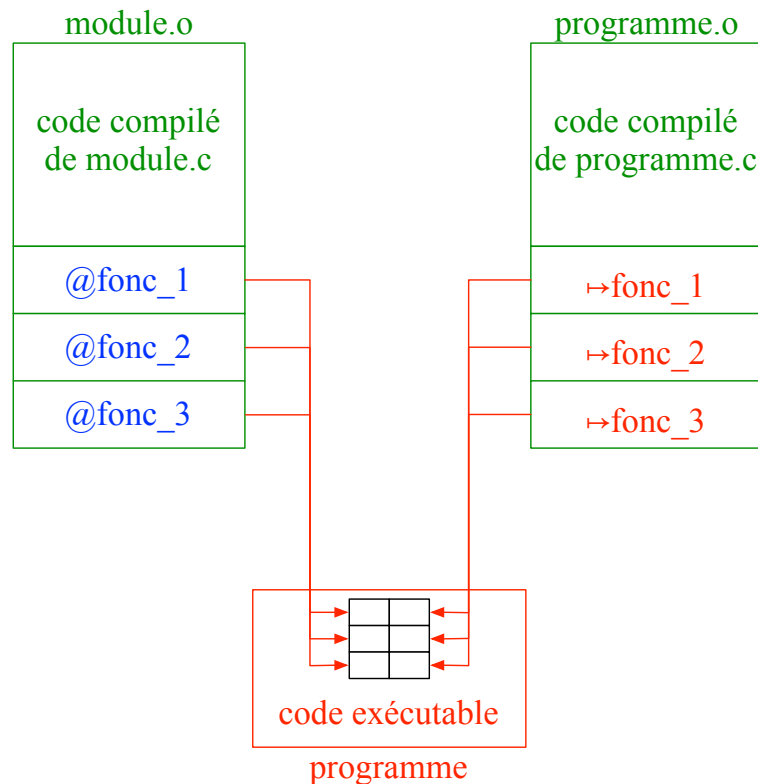
↪fonc\_3

# Fichiers Compilables (3)

- Fichier **exécutable**
  - pas d'extension
  - issu de l'édition de liens entre tous les objets des modules utilisés
    - ✓ appel automatique à `ld`
  - commande

```
$> gcc -o exécutable module1.o module2.o ... modulen.o
```

# Fichiers Compilables (4)

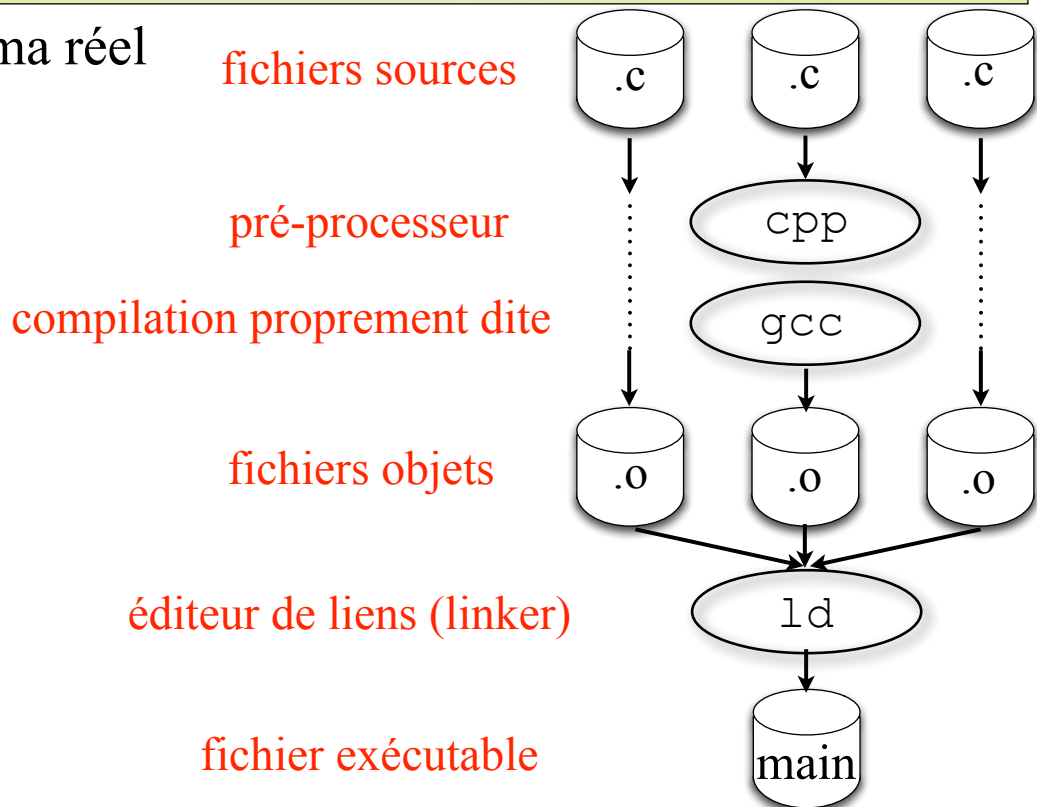


## Vision Globale

- 3 étapes distinctes s'enchaînent donc au cours de la compilation
  1. pré-traitement du code source
    - ✓ cpp
    - ✓ élimination des commentaires
    - ✓ inclusion de sous-fichiers
    - ✓ remplacement de macros
  2. compilation du code source pré-traité
    - ✓ gcc
    - ✓ un fichier objet par fichier source
  3. édition de liens
    - ✓ ld
    - ✓ fusion des fichiers objets pour créer le binaire exécutable

# Vision Globale (2)

- Schéma réel



## Options

- -I
  - augmenter la liste des répertoires dans lesquels chercher les fichiers d'en-tête
- -L
  - augmenter la liste des répertoires dans lesquels chercher les fichiers de bibliothèques
- -W
  - affichage des "warnings"
  - divers niveaux de "warnings"
- -Dvar=val
  - définition de variables du pré-processeur
  - -DDEBUG=2 ⇔ **#define** DEBUG 2
- -g
  - débogage

# Warnings

- Exemple

```
#include <stdio.h>

int main(){
    float *a;
    printf("%f\n", *a);

    return 0;
} //fin programme
```

- Compilation sans warning

```
$>gcc -o main warning.c
```

## Warnings (2)

- Compilation avec warning

```
$>gcc -o main -Wuninitialized warning.c
warning.c:6:21: warning: variable 'a' is uninitialized when used
here [-Wuninitialized]
    printf("%f\n", *a);
                    ^
warning.c:4:13: note: initialize the variable 'a' to silence this
warning
    float *a;
          ^
          = NULL
1 warning generated.
```

# Warnings (3)

- Les warnings sont donc
  - une aide pour le programmeur
  - lisible pour le programmeur
- En outre, les warnings permettent d'éviter les erreurs "silencieuses"

# Warnings (4)

- Warnings typiques

```
#include <stdio.h>

int oublie_retour(int x){
    int valeur = 0;
    valeur = valeur + x;
} //fin oublie_retour()

int *recupere_addr_temp(int valeur){
    return &valeur;
} //fin recupere_addr_temp()

int main(){
    int a = oublie_retour(3);
    printf("%d\n", a);

    int tableau[2] = {1, 2, 3};

    int b = 3;
    int *ptr = recupere_addr_temp(b);
    printf("%d\n", *ptr);

    int somme = 0;
    for(unsigned int k=5; k>=0; --k)
        somme += 3;

    return 0;
} //fin programme
```

```
$>gcc -o main -Wall -Wextra warning.c
warning.c:6:1: warning: control reaches end of
non-void function [-Wreturn-type]
} //fin oublie_retour()
^
warning.c:9:13: warning: address of stack memory
associated with local variable 'valeur' returned
[-Wreturn-stack-address]
    return &valeur;
           ^~~~~~
warning.c:16:29: warning: excess elements in
array initializer
    int tableau[2] = {1, 2, 3};
                        ^
warning.c:24:15: warning: comparison of unsigned
expression >= 0 is always true [-Wtautological-
compare]
    for(unsigned int k=5; k>=0; --k)
                        ^~ ^
warning.c:16:9: warning: unused variable
'tableau' [-Wunused-variable]
    int tableau[2] = {1, 2, 3};
```

# Warnings (5)

- Bonne pratique
  - toujours activer les warnings
    - ✓ dans le cadre du cours
      - `-Wall -W -Wmissing-prototypes`
- Quelques warnings utiles
  - `-Wswitch-default`
    - ✓ absence de branche par défaut
  - `-Wswitch-enum`
    - ✓ absence d'une valeur dans l'énumération ou présence d'une valeur en dehors de l'intervalle d'énumération
  - `-Wfloat-equal`
    - ✓ valeurs flottantes utilisées dans des comparaisons d'égalité
  - `-Wredundant-decls`
    - ✓ une variable est déclarée plusieurs fois avec la même portée
  - `-Winit-self`
    - ✓ variable non initialisée initialisée avec elle-même

# Warnings (6)

- Toujours activer un maximum de warnings
- Ne jamais se priver du travail du compilateur
- Plus d'informations
  - <https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>



# Efficacité

- Si un projet comprend plusieurs modules, comment recompiler après la modification de un (ou plusieurs) fichier(s)?
- La compilation "à la main" des fichiers modifiés est hasardeuse
- Créer un script shell?
  - recompilera tout à chaque fois
  - coûteux!
- Comment automatiser la recompilation en fonction des dépendances entre fichiers?
  - `make`

# Agenda

- Chapitre 1: Compilation
  - Compilation Multi-Fichiers
  - `make`
    - ✓ Généralités
    - ✓ Clauses
    - ✓ Evaluation
    - ✓ Variables
    - ✓ Compilation Séparée
  - Arborescence

# Généralités

- `make` est un programme permettant de n'effectuer que les traitements nécessaires à l'obtention d'un nouvel exécutable
  - recompile uniquement les fichiers modifiés
  - mise à jour des bibliothèques les concernant
  - édition des liens pour générer les exécutables
- Comment utiliser?

```
$> make [cible]
```

## Généralités (2)

- Afin de bien remplir sa fonction, `make` doit connaître
  - les dépendances entre fichiers
  - les traitements à appliquer
- Ces informations ne sont pas spécifiques aux programmes C
- `make` peut servir à tout type de traitement de mise à jour partielle
  - documents LaTeX multi-fichiers
  - compilation Java
  - ...

# Généralités (3)

- Les informations de dépendance sont contenues dans un fichier appelé *makefile* ou *Makefile*
  - `make` utilise le fichier *makefile* si il existe dans le répertoire courant
  - sinon, il utilise le fichier *Makefile*
- L'option `-f` de `make` permet de spécifier un autre nom de fichier, si besoin
  - pas conseillé

# Généralités (4)

- Un fichier *makefile* est composé de deux parties:
  - déclaration de variables
  - suite de clauses (ou règles)

# Généralités (5)

- Makefile

```
CC=gcc
LD=gcc
CFLAGS=--std=c99 --pedantic -Wall -W -Wmissing-prototypes
LDFLAGS=
EXEC=main
```

Déclaration de variables

```
all:$(EXEC)
```

règle principale

```
main: brol.o trol.o chmol.o
$(LD) -o main brol.o trol.o chmol.o $(LDFLAGS)
```

```
brol.o: brol.c brol.h
$(CC) -c brol.c -o brol.o $(CFLAGS)

trol.o: trol.c trol.h
$(CC) -c trol.c -o trol.o $(CFLAGS)

chmol.o: chmol.c chmol.h
$(CC) -c chmol.c -o chmol.o $(CFLAGS)
```

Règles

édition de liens

pré-processing  
et compilation

# Clauses

- Un fichier Makefile est un ensemble de clauses

cible: dépendances  
actions

- Le caractère ":" sépare la cible de la/les dépendance(s)
- Cible?
  - nom du fichier à mettre à jour ou simple label
- Dépendance?
  - liste de fichiers dont dépendent la cible

## Clauses (2)

- Comment savoir si une cible doit être reconstruite?
  - comparer la date de la cible et la date de chacune des dépendances
- Une fois les dépendances de la cible vérifiées et éventuellement mises à jour
  - les actions sont effectuées
  - si le fichier cible est moins récent que au moins l'un des fichiers de dépendances

**Si** date dernière modification dépendance(s) > date cible  
**Alors** mettre à jour la cible

## Clauses (3)

- Les dépendances sont soit
  - des fichiers sources existants "écrits" par l'utilisateur
    - ✓ cible: brol.c brol.h
  - des fichiers à construire
    - ✓ ils doivent avoir eux-même une règle indiquant comment les créer
    - ✓ cible: brol.o module\_exotique.o
  - vide
    - ✓ la cible ne dépend de rien et devra toujours être mise à jour
    - ✓ cible:

# Clauses (4)

- Syntaxe pour les actions
  - N lignes de commande pour reconstruire la cible
  - $N \geq 0$
- Si  $N > 0$ 
  - chaque ligne doit contenir le caractère de tabulation
  - une ou plusieurs commande séparées par ";"
  - exemple:  
✓ <tabulation> gcc -c brol.c -o brol.o
- Si  $N == 0$ 
  - on ne fait rien
  - on vérifie juste si la cible est à jour par rapport aux dépendances

# Clauses (5)

- Une ligne ne commençant pas par une tabulation indique une nouvelle cible
  - source d'erreurs!

```
all:$(EXEC)

main: brol.o trol.o chmol.o
❑$(CC) -o main brol.o trol.o chmol.o $(LDLAGS)

brol.o: brol.c brol.h
❑$(CC) -c brol.c -o brol.o $(CFLAGS)

trol.o: trol.c trol.h
❑$(CC) -c trol.c -o trol.o $(CFLAGS)

chmol.o: chmol.c chmol.h
❑$(CC) -c chmol.c -o chmol.o $(CFLAGS)
```

tabulation

# Evaluation

- Règle simple avec des fichiers sources

```
bro1.o: bro1.c bro1.h  
gcc -c bro1.c -o bro1.o
```

- Activation

```
$>make bro1.o
```

```
Si (date bro1.c > date bro1.o) ||  
    (date bro1.h > date bro1.o)  
Alors reconstruire bro1.o par gcc -c bro1.c -o bro1.o
```

# Evaluation (2)

- Règle avec des fichiers qu'il faut reconstruire

```
bro1: bro1.o  
gcc -o bro1 bro1.o
```

- Activation

```
$>make bro1
```

```
Si bro1.o à jour avec la règle précédente  
Alors  
    Si (date bro1.o > date bro1)  
    Alors reconstruire bro1 par gcc -o bro1 bro1.o
```

# Evaluation (3)

- Règle sans dépendance

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

- Activation

```
$>make clean
```

- La cible `clean` est toujours exécutée
  - ce type de cible est souvent utilisé pour remettre à zéro les dépendances entre fichiers

# Evaluation (4)

- Règle sans action

```
all: main  
  
main: brol.o trol.o chmol.o  
    $(LD) -o main brol.o trol.o chmol.o $(LDLAGS)  
  
brol.o: brol.c brol.h  
    $(CC) -c brol.c -o brol.o $(CFLAGS)  
  
trol.o: trol.c trol.h  
    $(CC) -c trol.c -o trol.o $(CFLAGS)  
  
chmol.o: chmol.c chmol.h  
    $(CC) -c chmol.c -o chmol.o $(CFLAGS)
```

- Activation

```
$>make all
```



# Evaluation (5)

- Vérification de la cible `main`
  - vérification de la cible `bro1.o`
    - ✓ vérification des fichiers `bro1.c` et `bro1.h`
  - vérification de la cible `trol.o`
    - ✓ vérification des fichiers `trol.c` et `trol.h`
  - vérification de la cible `chmol.o`
    - ✓ vérification des fichiers `chmol.c` et `chmol.h`
- On en revient à vérifier toutes les dépendances
  - la cible `all` est la racine de l'arbre des dépendances
  - souvent utilisé pour compiler tout un projet

# Evaluation (6)

- Par défaut, `make` évalue uniquement la première clause rencontrée et son arbre de dépendance
- On peut donner en paramètre à `make` le nom d'une autre cible
  - possibilité de définir des actions différentes
  - possibilité de définir des noms de cibles mnémotechniques sans interférences avec des cibles existantes au moyen de la cible `.PHONY`

# Evaluation (7)

- Exemples de cibles mnémotechniques

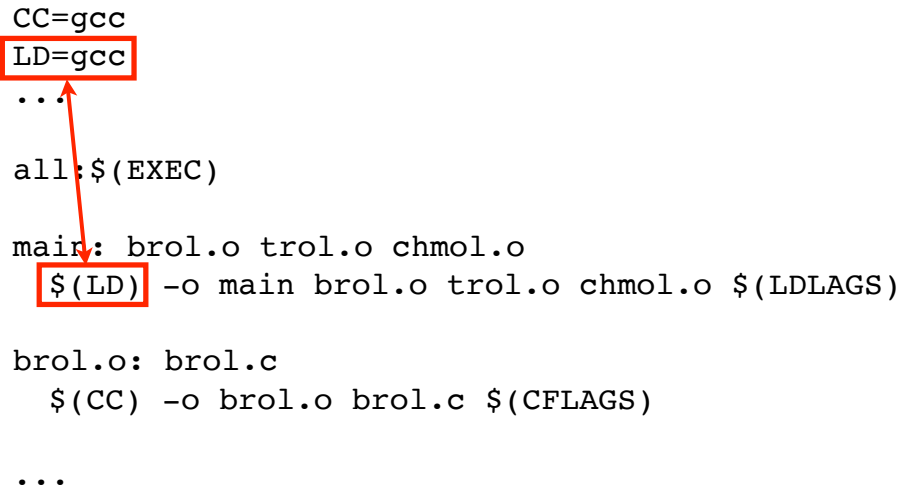
```
.PHONY: all clean archive  
  
all: $(EXEC)  
  
clean:  
    rm -f *.o main  
  
archive:  
    tar -zcvf projet.tar.gz *.h *.c README makefile  
  
bro1.o: ...
```

# Variables

- Avec make, il est possible d'utiliser des variables
  - analogues aux variables d'un script
  - permettent de factoriser les actions des règles
- Comment définir une variable?
  - NOM=valeur
- Comment utiliser une variable?
  - \$(NOM)

# Variables (2)

```
CC=gcc
LD=gcc
...
all:$(EXEC)
main: brol.o trol.o chmol.o
    $(LD) -o main brol.o trol.o chmol.o $(LDLAGS)
brol.o: brol.c
    $(CC) -o brol.o brol.c $(CFLAGS)
...
```



# Variables (3)

- Principales variables d'environnement gérées par make
  - CC: nom du compilateur C
  - CFLAGS: options de la première phase de compilation
  - LD: nom de l'éditeur de liens
  - LDFLAGS: options de l'édition de liens
  - CPP: nom du pré-processeur
  - CPPFLAGS: options du pré-processeur
  - RM: commande d'effaçage
- Il y en a d'autres

# Variables (4)

- Lors de l'évaluation d'une règle, `make` configure un certain nombre de variables utilisables au sein des actions
  - `$@`: le nom de fichier cible de la règle
  - `$*`: le nom de fichier cible, sans le suffixe
  - `$<`: le nom de la première dépendance
  - `$^`: la liste de toutes les dépendances
  - `$?`: la liste de toutes les dépendances qui sont plus récentes que la cible
    - ✓ utile, par exemple, pour la mise à jour de fichiers de bibliothèque

# Variables (5)

```
CC=gcc
LD=gcc

...

main: bro1.o trol.o chmol.o
    $(LD) -o main $^ $(LDLAGS)

bro1.o: bro1.c
    $(CC) -c $< -o $@ $(CFLAGS)

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

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

# Variables (6)

- Pour ne pas avoir à réécrire les mêmes actions pour chaque cible de même type, `make` offre un mécanisme de règles implicites
- Les règles implicites sont basées sur les suffixes

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

- Au lancement, `make` dispose de nombreuses règles prédéfinies pour compiler les fichiers en fichiers objets ou exécutables
  - exemple: `.c → .o`

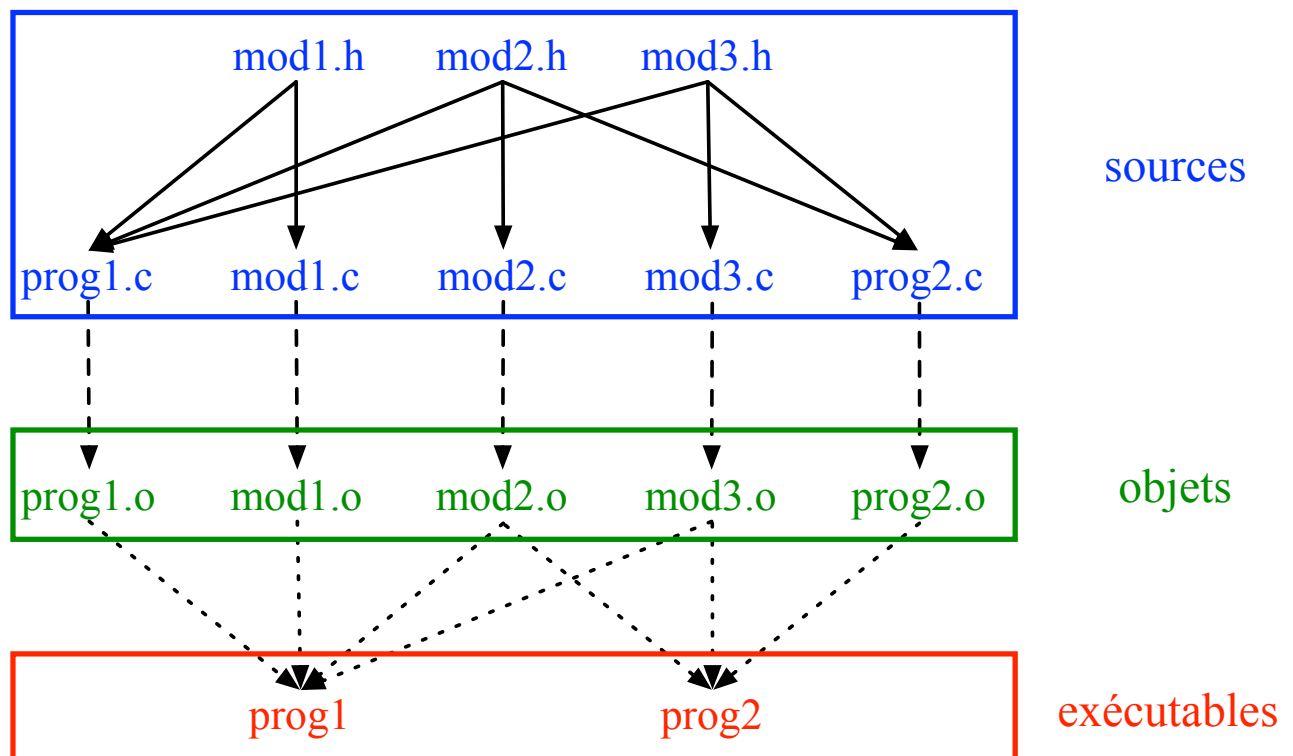
# Compilation Séparée

- 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
    - ✓ tout recompiler à chaque fois est inutile
- Le Makefile dépasse largement la compilation de programmes en C

# Compilation Séparée (2)

- Structure du projet
  - 2 programmes: `prog1.c` et `prog2.c`
  - 3 modules (sans `main()`): `mod1.{c,h}`, `mod2.{c,h}`, `mod3.{c,h}`
  - gestion de la documentation via Doxygen
    - ✓ cfr. Chapitre 4
- Les règles de dépendances sont les suivantes:
  - `prog1` utilise `mod1`, `mod2` et `mod3`
  - `prog2` utilise `mod2` et `mod3`
  - un fichier `.doc_conf` a été créé
    - ✓ génération de fichiers html via la commande `doxygen .doc_conf`

# Compilation Séparée (3)



# Compilation Séparée (4)

- Définition des variables

```
#-----  
# variables  
#-----  
CC=gcc  
LD=gcc  
CFLAGS=--std=c99 --pedantic -Wall -W -Wmissing-prototypes  
LDFLAGS=  
DOXYGEN=doxygen  
OBJ_P1=prog1.o mod1.o mod2.o mod3.o  
OBJ_P2=prog2.o mod2.o mod3.o
```

# Compilation Séparée (5)

- Cibles pour les programmes

```
#-----  
# programmes  
#-----  
all: prog1 prog2  
  
prog1: $(OBJ_P1)  
    $(LD) -o prog1 $(OBJ_P1) $(LDFLAGS)  
  
prog1.o: prog1.c  
    $(CC) -c prog1.c -o prog1.o $(CFLAGS)  
  
prog2: $(OBJ_P2)  
    $(LD) -o prog2 $(OBJ_P2) $(LDFLAGS)  
  
prog2.o: prog2.c  
    $(CC) -c prog2.c -o prog2.o $(CFLAGS)
```

# Compilation Séparée (6)

- Cibles pour les modules

```
#-----  
# modules  
#-----  
mod1.o: mod1.c mod1.h  
    $(CC) -c mod1.c -o mod1.o $(CFLAGS)  
  
mod2.o: mod2.c mod2.h  
    $(CC) -c mod2.c -o mod2.o $(CFLAGS)  
  
mod3.o: mod3.c mod3.h  
    $(CC) -c mod3.c -o mod3.o $(CFLAGS)
```

# Compilation Séparée (7)

- Cibles pour la documentation

```
#-----  
# Documentation  
#-----  
documentation:  
    $(DOXYGEN) .doc_conf
```

- Cibles pour des utilitaires

```
#-----  
# Utilitaires  
#-----  
clean:  
    rm -f *.o
```



# Agenda

- Chapitre 1: Compilation
  - Compilation Multi-Fichiers
  - make
  - Arborescence
    - ✓ Principe
    - ✓ Exemple

# Principe

- Si le projet est organisé en arborescence de répertoires, alors il faut une arborescence de Makefile
- Convention
  - un à la racine
    - ✓ Makefile père
  - un dans chaque répertoire
    - ✓ Makefile fils

# Principe (2)

- Makefile père
  - situé à la racine du répertoire
- Rôle?
  - définit les macros communes
  - invoque les Makefile fils
- Comment?
  - `cd Repertoire_Cible ; make <cible>`
- On peut
  - utiliser des inclusions
    - ✓ `include`
  - sélectionner certaines clauses
    - ✓ `ifeq, else, endif`

# Principe (3)

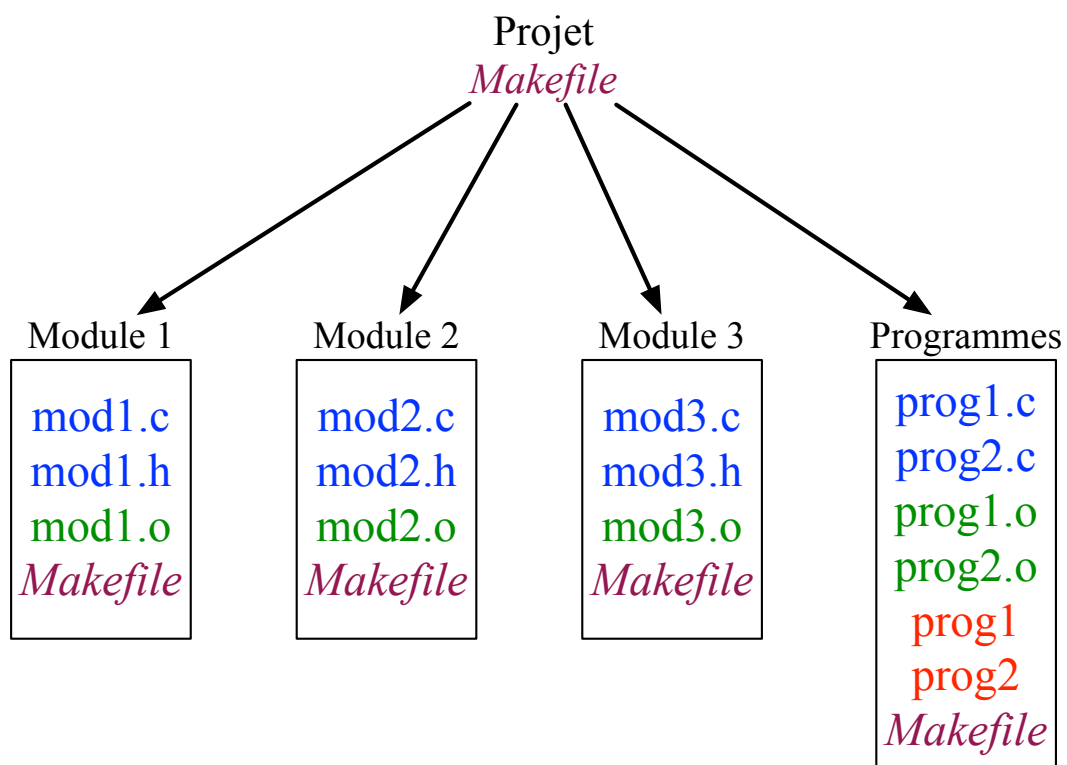
- Makefile fils
  - Makefile standard

# Exemple

- Exemple: 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 (2)



# Exemple (2)

- Déclaration des variables dans un fichier séparé
  - Makefile.compilation

```
CC=gcc
LD=gcc
CFLAGS=--std=c99 --pedantic -Wall -W -Wmissing-prototypes
LDFLAGS=
PROG1=prog1
PROG2=prog2
RM=rm
```

# Exemple (3)

- Makefile père

```
include Makefile.compilation
```

```
all:
```

```
  cd Module1; make all
  cd Module2; make all
  cd Module3; make all
  cd Programmes; make
```

```
clean:
```

```
  cd Module1; make clean
  cd Module2; make clean
  cd Module3; make clean
  cd Programmes; make clean
```

# Exemple (4)

- Makefile Module 1

```
include ../Makefile.compilation

#-----
# Modules
#-----
mod1.o: mod1.c mod1.h
    $(CC) -c mod1.c -o mod1.o $(CFLAGS)

#-----
# Gestion generale du projet
#-----
all: mod1.o

clean:
    rm *.o
```

- Makefile Programmes

```
include ../Makefile.compilation

#-----
# all
#-----
all: prog1 prog2
#-----
# Exécutables
#-----
prog1: ../Module1/mod1.o ../Module2/mod2.o
    $(LD) -o $(PROG1) ../Module1/mod1.o ../Module2/mod2.o $(LDLAGS)

prog2: ../Module2/mod2.o ../Module3/mod3.o
    $(LD) -o $(PROG2) ../Module2/mod2.o ../Module3/mod3.o $(LDLAGS)

#-----
# Gestion generale du projet
#-----
clean:
    rm *.o
```