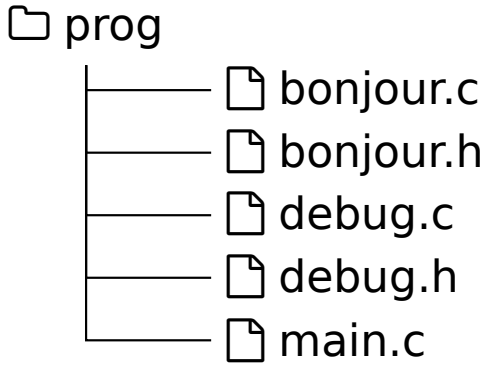
# Immersion 2023/24

## Makefile & make

D. Lienhardt

Septembre 2023

## Exemple




## Compilation manuelle

```
1    $ gcc -o prog *.c
```

## Limites de la compilation manuelle

- répétitions inutiles
- maintenance pratiquement impossible sur un gros projet

## debug.h

```
1  #ifndef TD_LIB_H
2  #define TD_LIB_H
3
4  #include <stdio.h>
5
6  #ifdef DEBUG
7
8  #define debug(x, ...) do { \
9    fprintf(stderr, "%s:%d:%s>␣",__FILE__,
         __LINE__,__func__); \
10   fprintf(stderr, x, ##__VA_ARGS__); \
11   fprintf(stderr, "\n"); \
12 } while (0)
13
14 #define trace() do { \
15   fprintf(stderr, "%s:%d:%s>␣Trace\n",
         __FILE__,__LINE__,__func__); \
16 } while (0)
17
18 #else
19
20 #define debug(x, ...)
21 #define trace()
22
23 #endif
24
25 #define MAX_ERR 100
26 void exit_error(const char *fmt, ...);
27
28 #endif
```

## debug.c

```
1  #include <errno.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <stdarg.h>
5
6  #include "debug.h"
7
8  void exit_error(const char *fmt, ...) {
9    char s[MAX_ERR];
10   va_list argp;
11   va_start(argp, fmt);
12   //vfprintf(stderr, fmt, argp);
13   vsnprintf(s, MAX_ERR, fmt, argp);
14   va_end(argp);
15   if (errno==0)
16     fprintf(stderr, "%s\n", s);
17   else
18     perror(s);
19   exit(EXIT_FAILURE);
20 }
```

## bonjour.h

```
1  #ifndef BONJOUR_H
2  #define BONJOUR_H
3
4  #define BJ_MAX 100
5
6  int bonjour(char *s);
7
8  #endif
```

## bonjour.c

```
1  #include <stdio.h>
2  #include <string.h>
3
4  #include "debug.h"
5  #include "bonjour.h"
6
7  int bonjour(char *s) {
8    trace();
9    char sp[BJ_MAX];
10   snprintf(sp, BJ_MAX, "Bonjour>␣%s", s);
11   printf("%s\n", sp);
12   return (int)strlen(sp);
13 }
```

## main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  #include "bonjour.h"
6  #include "debug.h"
7
8  int main(int argc, char *argv[]) {
9    int i;
10   trace();
11   int n=bonjour("Tout␣va␣bien");
12   return EXIT_SUCCESS;
13 }
```

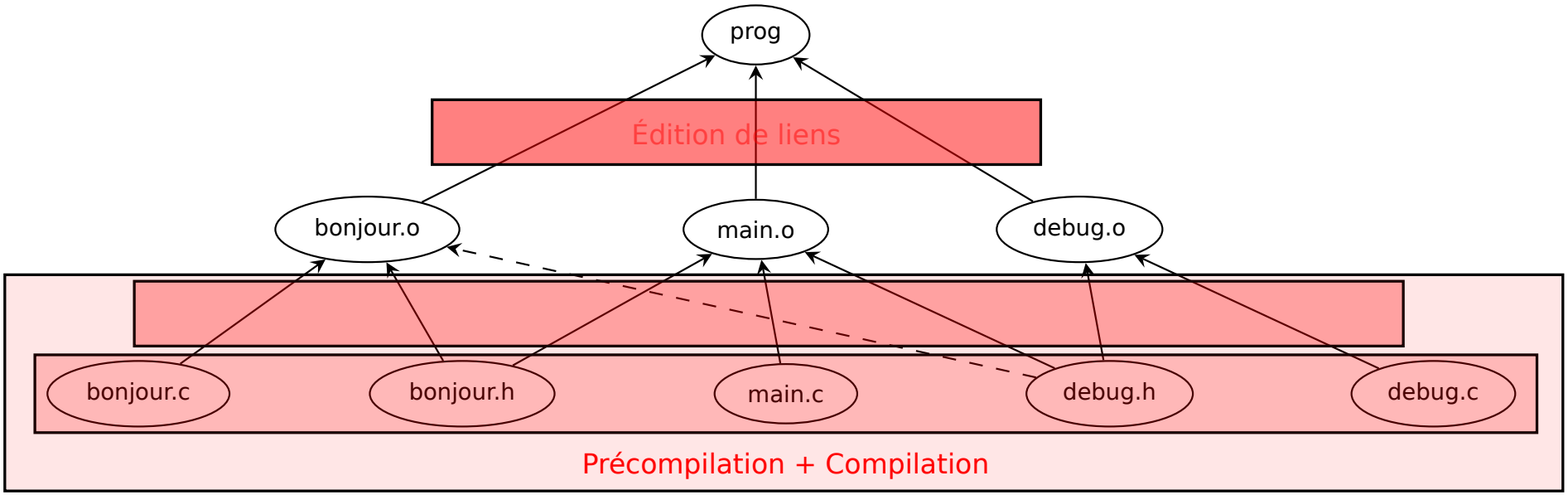
## Production de code exécutable (gcc)

- La phase de prétraitement (*preprocessing*) est assurée par le préprocesseur : il s'agit d'un traitement de texte destiné à la génération du code source net à compiler (`#include`, `#define`, ...) qui ne concerne que les sources bruts (`.c` et `.h`) créés par l'utilisateur
- La compilation : génération d'un code objet (fichier `.o`) pour chaque fichier source net (résultat de la phase précédente)
- L'édition de lien : fusion des objets en un exécutable unique avec résolution des adresses

La commande gcc réalise les trois phases par défaut (`.c` et `.o` anonymes)
Pratiquement, gcc n'est utilisé séparément que deux fois

- Prétraitement et compilation (`.c` anonymes et `.o` nommés)
- Édition de lien

Graphe de dépendances (graphe orienté acyclique)




Couple (`make`, `Makefile`)

- Le fichier `Makefile` décrit les dépendances et les actions à réaliser
- La commande `make` doit résoudre le problème du tri topologique (

## Composition d'un fichier `Makefile`

Un fichier `Makefile` est composé d'un ensemble de règles respectant la syntaxe :

```
cible : dépendances
  commande
  ...
  commande
```

- Chaque règle est identifiée par un identifiant unique (qui peut désigner une ou plusieurs cibles)
- `make` traite la 1re règle rencontrée ou la règle désignée (argument de `make`)
- Pour traiter une règle, les dépendances sont d'abord évaluées
- Si une dépendance est elle-même l'identifiant d'une autre règle, cette dernière sera préalablement traitée (tri topologique)
- Après évaluation des dépendances, les commandes sont exécutées si
  - il n'existe aucun fichier du nom de la cible
  - un fichier portant le nom d'une des dépendances est plus récent que le fichier cible

## Un premier `Makefile` (statique et brut) de notre exemple

```
1  prog: bonjour.o debug.o main.o
2    gcc -o prog bonjour.o debug.o main.o
3
4  bonjour.o: bonjour.c bonjour.h debug.h
5    gcc -c -o bonjour.o bonjour.c
6
7  debug.o: debug.c debug.h
8    gcc -c -o debug.o debug.c
9
10 main.o: main.c bonjour.h debug.h
11   gcc -c -o main.o main.c
```

Quelques défauts

- Maintenance lourde (redondances)
- Compilation comme tâche unique (suppression des fichiers intermédiaires et autres non prévues)

## Alléger l'écriture par des variables

- Variables définies par l'utilisateur
  - Déclaration d'une variable
    - `nom=valeur` (*dynamique*)
    - `nom:=valeur` (*statique*)
    - `nom?=valeur` (*defaut*)
  - Utilisation d'une variable `$(nom)`
- Variables internes (dans une commande)
  - `$@` le nom de la cible
  - `$<` le nom de la première dépendance
  - `$^` la liste énumérée des dépendances (sans duplication)
  - `$+` la liste énumérée des dépendances
  - `$?` la liste énumérée des dépendances plus récentes que la cible
  - `$*` le nom de la cible sans suffixe

## Ajout de règles supplémentaires

- suppression des fichiers intermédiaires
- autres à venir...

## Première évolution du `Makefile`

```
1  CC=gcc
2  CPPFLAGS= -DDEBUG
3  CFLAGS= -std=c99
4  LDFLAGS=
5  EXEC=prog
6  all: $(EXEC)
7
8  prog: bonjour.o debug.o main.o
9    $(CC) -o $@ $^ $(LDFLAGS)
10
11 bonjour.o: bonjour.c bonjour.h debug.h
12   $(CC) -c $(CPPFLAGS) $(CFLAGS) -o $@ $<
13
14 debug.o: debug.c debug.h
15   $(CC) -c $(CPPFLAGS) $(CFLAGS) -o $@ $<
16
17 main.o: main.c bonjour.h debug.h
18   $(CC) -c $(CPPFLAGS) $(CFLAGS) -o $@ $<
19
20 clean:
21   rm -f *.o
22
23 cleanall: clean
24   rm -f $(EXEC)
```

Répétition des lignes de compilation (12, 15 et 18) : règle implicite

## Évolution du `Makefile` après règles implicites et dépendances

```
1  CC=gcc
2  CPPFLAGS= -DDEBUG
3  CFLAGS= -std=c99
4  LDFLAGS=
5  EXEC=prog
6  all: $(EXEC)
7
8  prog: bonjour.o debug.o main.o
9    $(CC) -o $@ $^ $(LDFLAGS)
10
11 %.o: %.c
12   $(CC) -c $(CPPFLAGS) $(CFLAGS) -o $@ $<
13
14 # Dépendances
15 bonjour.o: bonjour.c bonjour.h debug.h
16 debug.o: debug.c debug.h
17 main.o: main.c bonjour.h debug.h
18
19 .PHONY: clean cleanall
20
21 clean:
22   rm -f *.o
23
24 cleanall:
25   rm -f *.o $(EXEC)
```

`gcc`, couteau suisse de la compilation

- `gcc` capable de produire des éléments de couverture de code (cf tests)
- `gcc` capable de produire des éléments de débogage

Et les dépendances ?

- `gcc` est bien sûr aussi capable de fournir les dépendances (options de prétraitement `-MMD -MP`)
- Chaque fichier `%.c` donnera lieu à un nouveau fichier intermédiaire `%.d` lequel (au format d'un Makefile) décrira les dépendances du fichier `%.o`
- Il n'y aura plus qu'à inclure les fichiers `%.d` dans le Makefile par une clause `include` (voire `-include`)

## Fonctions de listes et autres

- Générer une liste des fichiers sources

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

- Générer une liste induite d'après les extensions

```
deps := $(SRC:.c=.d)
```

ou par fonction de substitution

```
deps := $(patsubst %.c,%.d,$(SRC))
```

- inclusion des dépendances

```
include $(deps)
```

- exécution de `make` dans un Makefile (danger récursion)

```
$(MAKE) -C $(subdir)
```

Travail demandé dans le cadre du projet

Étant placé à la racine du répertoire de projet

- la compilation du projet doit être obtenue par la simple commande

```
$ make
```

- l'exécution du programme doit être obtenue par la ligne de commande conforme aux spécifications

```
$ ./hex [options]
$ ./hex -s pppp
$ ./hex -c hh.hh.hh.hh:pppp
$ ./hex -l bleu
$ ./hex -l rouge
```

- la production et l'exécution des tests sera obtenue par la commande

```
$ make tests
```

- la production de la documentation sera obtenue par la commande

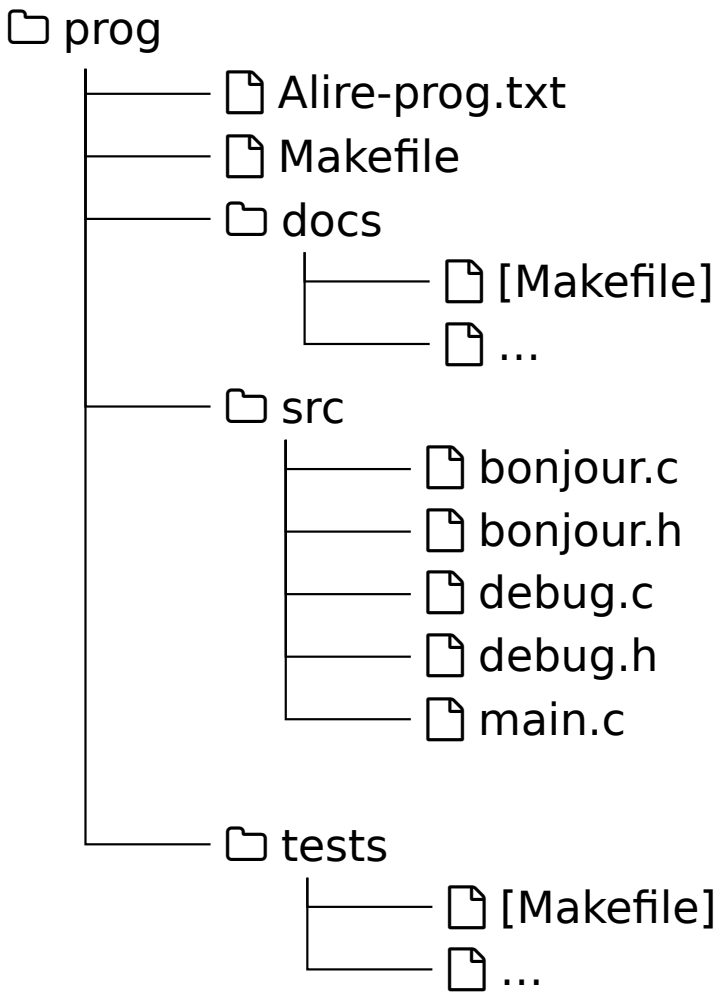
```
$ make docs
```

**Obligatoire**

Les alinéas 1 et 2 sont obligatoires

## Structure de projet suggérée

### Arborescence




### Alire-prog.txt

```
+-----------------------------------------------
| Compilation
+-----------------------------------------------

Au niveau le plus haut (*)
  Compilation, produit l'exécutable ./prog
    $ make
  Tests : compile et exécute les tests sous
  ./tests
    $ make tests
  Docs : production des docs sous ./docs
    $ make docs

+-----------------------------------------------
| Utilisation
+-----------------------------------------------

$ ./prog options (*)

(*) Respect des consignes
```

Pour aller plus loin...

[1] Peter Miller. *Recursive Make Considered Harmful*. (Moteur de recherche : +pdf). 2002.

[2] Richard M. Stallman, Roland McGrath et Paul D. Smith. *GNU Make*. 4.4. (`https://www.gnu.org/software/make/manual/make.pdf`). Free Software Foundation, 2022. isbn : 1-882114-83-3.