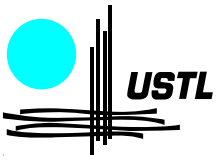

Licence d'informatique
Module de Pratique du C

TP introductif de pratique du C

Philippe MARQUET

Révision d'octobre 2007

Ce premier TP introductif à la programmation en C va vous familiariser avec :
- la syntaxe du langage ;
- la mise en œuvre du compilateur ;
- le préprocesseur `cpp` ;
- l'utilisation d'un « débogueur/dévermineur/metteur au point » ;
- la compilation séparée et l'utilisation de `make` ;
- divers outils relatifs au dévéloppement de programmes en C.

Quelques unes des questions du présent sujet de TP sont inspirées d'un TP proposé par Christian QUEINNEC sur son site VideoC, `http://videoc.lip6.fr`

Vous pouvez copier/coller les sources C et les commandes du présent sujet depuis la version en ligne accessible à `http://www.lifl.fr/~marquet/ens/pdc/tpic`.

1 Edition de fichers source C

De nombreux éditeurs sont disponibles sur les machines des salles TP. Une bonne pratique est de choisir un outils et de se l'approprier en en faisant usage systématiquement et en en explorant toutes les fonctionnalités ; soyez curieux... Nous vous recommandons l'usage d'`emacs` .

Le mode C d'emacs offre des facilités pour programmer en C. Ce mode est activé quand on édite un fichier dont le nom se termine par `.c` ou `.h`.

Exercice 1
Regardez le menu ajouté à `emacs` quand vous éditez un source C. Essayez par exemple en sauvegardant le code C suivant (volontairement mal indenté) dans un fichier `foo.c` puis en indentant correctement le code :

```
/* ------------------------------
   Edition d'un fichier source C
   ------------------------------------------------------------ */

 const int MAX=12, MIN=0;
          extern int foo, bar, gee;

    int
                        main()
{

if (foo == MAX) {
  if gee = MIN;
  } else {
  gee = MAX;
bar = foo + gee;
}
```

```
bar *= foo - gee;
}
```

Les commandes suivantes sont des plus utiles (**C-** désigne la touche <Ctrl>, **M-** désigne la touche <Meta>, c'est-à-dire <ESC> ou <Echap> ou <Alt>) :

TAB Aligne le texte de la ligne correctement selon le niveau d'imbrication

M-; Crée un commentaire ;

M-x compile Lance une compilation dans le répertoire courant en appelant par défaut `make` ;

C-x' Se place sur la prochaine erreur de compilation ;

M-x gdb Invoque le debogueur, par défaut `gdb`.

Exercice 2
Essayez ces commandes lors de la suite de votre TP.

2 Compilation en plusieurs phases

C est un langage compilé. Cela signifie qu'un programme C est décrit par un fichier texte appelé fichier source. Ce fichier n'est pas exécutable par le microprocesseur, il faut le traduire en langage machine. Cette opération est effectuée par un programme appelé compilateur.

La compilation d'un programme C se décompose en 4 phases successives :

1. Le *traitement par le préprocesseur* : le fichier source est analysé par un programme appelé préprocesseur qui effectue des transformations purement textuelles (remplacement de chaînes de caractères, inclusion d'autres fichiers source, etc.).
2. La *compilation* : au cours de cette étape, le fichier engendré par le préprocesseur est traduit en assembleur, c'est à dire en une suite d'instructions qui sont chacunes associées à une fonctionnalité du microprocesseur (faire une addition, une comparaison, etc.).
3. L'*assemblage* : cette opération transforme le code assembleur en un fichier binaire, c'est-à-dire en instructions directement compréhensibles par le processeur. Le fichier produit par l'assemblage est appelé fichier objet `.o`).
4. L'*édition de liens* : un programme est souvent séparé en plusieurs fichiers source (ceci permet d'utiliser des librairies de fonctions standard déjà écrites comme les fonctions d'affichage par exemple). Une fois le code source assemblé, il faut donc lier entre eux les différents fichiers objets. L'édition de liens produit alors un fichier exécutable.

La commande `gcc` invoque tour à tour ces différentes phases. Des options de `gcc` permettent de stopper le processus de compilation après chaque des phases.

Nous allons illustrer cela sur le programme suivant (`phases.c`) :

```
/* ------------------------------
   Mon premier programme
   ------------------------------------------------------------ */

#include <stdio.h>
#include <stdlib.h>

#define MAX     12

int foo = 3;
const int bar = 5;

extern int gee;
```

```
int
main()
{
    int bzu;             /* declaration et definition de bzu */

    bzu = MAX;
    printf("Hello : %d\n", foo + bar + gee + bzu);

    exit(EXIT_SUCCESS);
}
```

Préprocessing

Exercice 3
Vérifiez dans la page de manuel de `gcc` le sens de l'option `-E`.

Exercice 4
Produisez le résultat du prétraitement par le préprocesseur sur le programme `phases.c` dans un fichier `phases.i`.

Reportez vous éventuellement au TP précédent pour vous remémorer l'utilisation des redirections des entrées/sorties des commandes.

Exercice 5
Comparez le contenu des fichiers `phases.c` et `phases.i`. En particulier

1. Combien de lignes comporte chacun des fichiers `phases.c` et `phases.i` ?
2. Qu'est devenu la ligne
   ```
   #include <stdio.h>
   ```
3. Que sont devenus les commentaires ?
4. Quel traitement a été réalisé sur la ligne
   ```
   bzu = MAX;
   ```
   Vous pouvez vérifiez cela en invoquant la commande « Macro expand region » (`C-c C-e`) du menu C d'`emacs` sur cette ligne.

Génération de code assembleur

Exercice 6
Vérifiez dans la page de manuel de `gcc` le sens de l'option `-S` et produisez le code assembleur correspondant à `phases.c`.

Exercice 7
Dans ce code assembleur, identifiez les différentes sections `.data`, `.rodata`, et `.text`.

1. Quels symboles sont associés à la section `.data` ? À quoi correspond cette section ?
   Vérifiez en ajoutant une déclaration au programme `phases.c`.
2. Quels symboles et valeurs sont associés à la section `.rodata` ? À quoi correspond cette section ?
   Vérifiez en modifiant `phases.c` pour ajouter un élément à cette section.
3. Quel est le point d'entrée de ce programme ?

Référez vous éventuellement à vos manipulations de l'assembleur `gas` vu dans l'UE « Architecture élémentaire des ordinateurs (info 202) ».

Génération du fichier objet

Exercice 8
Trouvez comment indiquer à `gcc` de générer le fichier objet `phases.o` à partir de `phases.c`.

Exercice 9
Comparer les résultats de la commande `nm` sur ce fichier `.o` avec ce que vous aviez identifié comme section dans le code assembleur.

Exercice 10
1. Qu'indique `nm` pour les symboles `printf`, `exit`, et `gee` ?
2. Où ces symboles sont-ils définis ? Quel programme est en charge de rechercher ces symboles ?
3. Qu'est ce qui change dans le résultat de `nm` si on supprime l'utilisation de `gee` dans l'instruction `printf` du fichier `phases.c` ?

Exercice 11
Comment expliquez vous le résultat de la commande `strings` sur ce fichier `.o` ?

En conclusion

Ces manipulations vous ont permis d'aborder le processus de compilation d'un programme C.

Ces manipulations sont représentatives de l'activité d'un programmeur :
- allers et retours entre un éditeur de textes et un shell ;
- consultation de pages de manuel ;
- consultation de documentation (web...).

Soyez à l'aise et efficaces avec vos outils !

Concernant les différentes phases de compilation :
- bien souvent on se contente d'appeler `gcc` qui enchaîne automatiquement toutes les phases de compilation jusqu'à produire un fichier exécutable ;
- on vérifie parfois le résultat fourni par le préprocesseur quand on a un doute sur une ligne de code comportant de nombreuses substitutions de macros ;
- la génération de code assembleur est parfois utilisée pour comparer le code généré et donc l'efficacité de deux versions de code C ;
- la production de fichiers objets est, quand à elle, très courante et forme la base de la compilation séparée. Nous y reviendrons.

3 Première compilation et exécution

Exercice 12
Écrivez un programme qui calcule la factorielle de 10, et affiche le résultat. On définira une fonction

```
unsigned int
factorielle (unsigned int n)
{
  unsigned int i = 1, res = 1;

  while (i  n)
   res = res * i++;
  return res;
}
```

qui calcule la factorielle de `n` de façon itérative. On placera le source de ce programme dans le fichier source `factorielle.c`.

Exercice 13
Compilez ce programme (le fichier source) par la commande

```
% gcc -g -Wall -Werror -ansi -pedantic -o factorielle factorielle.c
```

- on utilise le compilateur C du projet GNU, `gcc` ;
- les options `-ansi` et `-pedantic` précisent que l'on produit des programmes C au standard ANSI C et ce de manière stricte. **Ces options seront systématiquement utilisées pour tous vos développements**, au moins dans le cadre de ce module ;
- de même, vous utiliserez systématiquement les options `-Wall` (signaler tous les warnings) et `-Werror` (considérer les warnings comme des erreurs) ;
- l'option `-g` du compilateur C permet de conserver des informations utiles au déverminer ;
- l'option `-o` permet de spécifier le nom du fichier exécutable, (`factorielle` ici, `a.out` par défaut).

Si il n'y a pas d'erreur, le compilateur n'affiche rien et génère l'exécutable `factorielle`. Sinon, le compilateur affiche la liste des erreurs qu'il a détectées. Par exemple

```
% gcc -g -Wall -Werror -ansi -pedantic -o factorielle factorielle.c
factorielle.c: In function 'factorielle':
factorielle.c:6: parse error before 'n'
cc1: warnings being treated as errors
factorielle.c: At top level:
factorielle.c:11: warning: return-type defaults to 'int'
```

indique que dans la fonction `factorielle` du fichier source `factorielle.c`, il y a une erreur. C'est une erreur de syntaxe à la ligne 6 (`while (i n)`) où il manque un opérateur entre `i` et `n`. La seconde erreur est une admonestation (warning) à la ligne 11 (`main()`), où le type retourné par la fonction `main` a été fixé par défaut à `int`. Lisez donc toujours bien les messages d'erreurs.

Exercice 14
Corrigez le programme `factorielle.c`. Exécutez ensuite ce programme.

```
% factorielle
362880
```

4 Déverminage d'un programme

Un déverminer (ou metteur au point ou débogueur) permet d'exécuter un programme en mode pas à pas et de visualiser l'ensemble de la mémoire du processus correspondant au programme en cours d'exécution. La plupart des processeurs possède un mode pas à pas dans lequel ils appellent automatiquement une routine après l'exécution de chaque instruction machine. Les déverminers utilisent ce mode pour exécuter un programme :

- instruction par instruction (`stepi`, `nexti`) ;
- ligne de code source par ligne de code source (`step`, `next`) ;
- point d'arrêt par point d'arrêt (`continue`, `cont`).

C'est ce dernier mode que l'on utilise le plus fréquemment :

- on place un point d'arrêt (`break`) dans le code source à l'endroit où l'on suspecte que se trouve l'erreur ;
- on démarre l'exécution du programme (`run`) jusqu'à ce point d'arrêt ;
- puis on exécute pas à pas en examinant le comportement du programme.

On peut également examiner le contenu des variables (`print` et `display`) ou même la valeur d'une expression quelconque du langage C.

La pile des appels de fonction est aussi visible (`backtrace`), ainsi que les cadres d'appel de chaque fonction (`frame` et `info frame`).

Le dévermineur que vous allez utiliser est `gdb` (Gnu DeBugger) et plusieurs interfaces graphiques à `gdb` sont disponibles, dont `xxgdb` et `ddd`, ce dernier étant le moins austère.

Exercice 15

1. Lancez le dévermineur et chargez votre programme.
2. Placez un point d'arrêt avant l'appel à la fonction `factorielle()`.
3. Exécutez alors le programme en mode pas à pas (`step` pour entrer dans la fonction, `next` saute les appels de fonction).
4. Affichez les valeurs des variables `res` et `i` (`display`) : leurs valeurs sont quelconques car elles n'ont encore pas été initialisées. Les variables locales sont en effet allouées automatiquement dans la pile, qui à cet instant de l'exécution peut contenir n'importe quoi.
5. En continuant à exécuter en mode pas à pas, vous verrez les variables `res` et `i` évoluer au cours des itérations de la boucle.

Exercice 16

Relancez l'exécution du programme (`run`) avec le même point d'arrêt. Modifier la valeur de `n` dans la fonction `main()` avec l'instruction `set` (via le bouton *set*) ou par :

```
(gdb) set n=12
```

et terminez son exécution (`cont`). Le programme a bien calculé la factorielle de 12, et non pas de 10. Recommencez avec `n=13`, et vérifiez que le résultat n'est pas égal à 13 fois le précédent. Les entiers de type `int` ne font bien que 32 bits sur cette machine.

5 Compilation conditionnelle et préprocesseur

Le préprocesseur permet de sélectionner des zones de code à compiler ou à ne pas compiler selon différentes conditions. Il est ainsi possible de tester l'existence de macros avec les directives `#ifdef/#endif`, et d'inclure ou non le code compris entre ces deux directives. On peut aussi tester la valeur d'une macro avec la directive `#if/#endif`. Ceci s'avère très utile pour fournir des programmes pouvant être compilés sur différentes architectures. Par exemple :

```
#if ARCH = i386
/* code pour un intel i386 */
#endif
#if ARCH = alpha
/* code pour un DEC Alpha */
#endif
```

Exercice 17

Renommez la fonction `factorielle()` en `factorielle_iterative()`. Écrivez la fonction

```
unsigned int factorielle_recursive (unsigned int n);
```

qui calcule la factorielle de `n` de façon récursive. Placez les corps de ces fonctions dans une directive `#ifdef/#endif` de la manière suivante :

```
#ifdef RECURSIVE
unsigned int
factorielle_recursive (unsigned int n)
{
  ...
}

unsigned int
factorielle (unsigned int n)
{
```

```
  return factorielle_recursive (n);
}
#else
unsigned int
factorielle_iterative (unsigned int n)
{
...
}

unsigned int
factorielle(unsigned int n)
{
  return factorielle_iterative (n);
}
#endif
```

De cette manière, si la macro `RECURSIVE` est définie, seule la fonction `factorielle_recursive()` sera définie, et l'appel à la fonction `factorielle()` sera un appel à `factorielle_recursive()`. Sinon seule `factorielle_iterative()` est définie, et un appel à `factorielle` sera un appel à `factorielle_iterative()`.

La macro `RECURSIVE` peut être soit définie dans le fichier source (`#define RECURSIVE`), soit définie au moment de la compilation grâce à l'option `-D nom=definition` du compilateur.

Exercice 18
Vérifiez le travail du préprocesseur `cpp` en demandant au compilateur de ne faire que la phase de préprocessing :

```
% gcc -E -P factorielle.c
```

Vous voyez apparaître les déclarations de types, variables et fonctions standard, puis celles contenues dans `<stdio.h>`, et enfin la définition de `factorielle_iterative()`, `factorielle()` et `main()`.

Exercice 19
Recommencez en définissant la macro `RECURSIVE` :

```
% gcc -E -P -D RECURSIVE factorielle.c
```

C'est bien maintenant la version récursive qui est définie.

Exercice 20
Compilez la version récursive pour étudier son comportement sous dévermineur (par exemple `xxgdb`, vous pouvez préférer `ddd`) :

```
% gcc -D RECURSIVE -g -Wall -Werror -ansi -pedantic -o factorielle factorielle.c
% xxgdb factorielle
```

1. Placez un point d'arrêt avant le `return` de `factorielle_recursive()` et lancez l'exécution (`run`).
2. Puis avancez de quelques appels récursifs (`step`).
3. Vous pouvez examiner la pile des appels de fonctions avec la commande `bt`, et vous placer dans n'importe quel cadre d'appel par la commande `frame n` où `n` est le numéro du cadre choisi (à gauche dans l'affichage fait par `bt`). Par exemple,

```
(xxgdb) print n
$1 = 4
(xxgdb) frame 3
#3  0x804840d in factorielle_recursive (n=7) at factorielle.c:5
(xxgdb) print n
$2 = 7
```

6 Compilation séparée et utilisation de l'utilitaire `make`

La commande `make` permet de déterminer automatiquement quelles parties d'un programme sont à recompiler et génère les commandes pour ce faire. Il est très utile pour la compilation séparée, où un programme est constitué de plusieurs fichiers sources.

Exercice 21
Séparez votre programme en deux fichiers sources `fbib.c` et `fact.c`. Le premier ne contient que les définitions des fonctions `factorielle_iterative()`, `factorielle_recursive()` et `factorielle()`. Le second ne contient que la fonction `main()`. Créez un fichier `fbib.h` contenant la déclaration de la fonction `factorielle` :

```
extern unsigned int factorielle(unsigned int);
```

et incluez ce fichier dans `fact.c` :

```
#include <stdio.h>
#include "fbib.h"
...
```

Remarquez l'utilisation des doubles quotes pour indiquer au compilateur de chercher le fichier d'entêtes `fbib.h` dans le répertoire courant en premier. Sans utiliser `make`, pour compiler le programme, il faut exécuter les commandes suivantes:

```
% gcc -c -Wall -Werror -ansi -pedantic fbib.c
% gcc -c -Wall -Werror -ansi -pedantic fact.c
% gcc -Wall -Werror -ansi -pedantic -o fact fact.o fbib.o
```

L'option `-c` de `gcc` permet de spécifier au compilateur de ne pas faire l'édition de liens, et de ne générer que les fichiers objets `.o`.

Exercice 22
Créez un fichier `Makefile` (ou `makefile`) avec le contenu suivant :

```
CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic
CFLAGS  += -g

fact: fact.o fbib.o
        $(CC) $(CFLAGS) -o fact fact.o fbib.o
fact.o: fact.c
        $(CC) -c $(CFLAGS) fact.c
fbib.o: fbib.c
        $(CC) -c $(CFLAGS) fbib.c
```

Les lignes correspondantes aux commandes à exécuter commencent par une *tabulation* (et non une série d'espaces...). Lancez la compilation par la commande `make` (ou `make fact` si la première règle de dépendance ne concerne pas `fact`). Les trois compilations sont automatiquement exécutées. En modifiant la date du fichier `fbib.c`, et en recompilant,

```
% touch fbib.c
% make
```

seule la compilation de `fbib.c` et l'édition de liens sont exécutées, le fichier `fact.c` étant resté inchangé.

Exercice 23
En fait, `make` connaît déjà un ensemble de règles de dépendance par défaut, et seule la première dépendance suffit. L'option `-p` de `make` liste la base de données (variables et règles de dépendance) utilisée. Pour vous en convaincre, essayez de supprimer les 4 dernières lignes de `Makefile`.

Dans un `Makefile`, on place aussi une entrée `clean` pour faire le ménage des fichiers intermédiaire et une entrée `realclean` qui ne conserve que le strict nécessaire. Ces entrées ne correspondent pas à des fichiers à construire, on le précise à `make` par la directive `.PHONY` :

```
CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic
CFLAGS  += -g

fact: fact.o fbib.o
        $(CC) $(CFLAGS) -o fact fact.o fbib.o

.PHONY: clean realclean
clean:
        $(RM) factorielle.o main.o
realclean : clean
        $(RM) factorielle
```

Partisan du moindre effort (et de la moindre erreur), on peut rassembler l'ensemble des fichiers `.o` dans une variable `OBJ` et utiliser la variable automatique `$@` :

```
CC      = gcc
CFLAGS  = -Wall -Werror -ansi -pedantic
CFLAGS  += -g

OBJ     = fbib.o fact.o

fact: $(OBJ)
        $(CC) $(CFLAGS) -o $@ $(OBJ)

.PHONY: clean realclean
clean:
        $(RM) $(OBJ)
realclean : clean
        $(RM) fact
```

On termine alors une séance de TP par :

```
% make realclean
rm -f fbib.o fact.o
rm -f fact
```

7 Autres outils de développement

Devant les difficultés et pièges du langage C, d'autres outils sont aussi à considérer.

`lint`

`lint`, dont une version moderne est `splint` (Secure Programming Lint, installé sur le réseau enseignement et disponible à `http://www.splint.org/`), fournit nombre d'avertissements sur votre code C.

Exercice 24
Essayez la commande `splint` sur les sources écrits lors de ce TP.

gprof

`gprof` (GNU profiler) analyse les performances d'une exécution d'un programme. Le programme C doit avoir été instrumenté lors de la compilation (option `-pg` du compilateur). L'exécution du programme produira un fichier `gmon.out`. L'exploitation de ce fichier par `gprof` permet de visualiser l'arbre des appels des fonctions du programme, le pourcentage de temps CPU passé dans chacune des fonctions, etc.

nm

`nm` liste les symboles présents dans un fichier objet (`.o`) ou un exécutable. Il est ainsi possible d'identifier les variables définies et exportées, les fonctions définies, les fonctions utilisées sans être définies... dans un fichier objet.

8 Programmez maintenant

Il s'agit, pour terminer, de produire un programme `tronque` qui coupe les lignes d'un texte lu sur l'entrée standard qui comportent plus de 60 caractères. La coupure ne doit se faire que sur la première limite de mots qui suit le 60e caractère.

Exercice 25
Définissez précisément ce qu'est un mot.

Exercice 26
Développez une version C du programme `tronque`. Cela implique de produire

1. un algorithme solution du problème.
2. l'architecture du programme : découpage en fichiers, fonctions...
3. un fichier `Makefile` basé sur ce découpage.
4. le codage en C du programme.

À titre indicatif, vous pouvez écrire une commande `tronque` qui lit les caractères de l'entrée standard un à un à l'aide de `getchar()` et les traite au fur et à mesure.

Exercice 27
Définissez un jeu d'essai pour le programme `tronque`. Un jeu d'essai n'est pas un simple fichier sur lequel vous allez essayer le comportement de votre programme, mais un ensemble de fichiers au contenu *judicieusement* choisis pour tester le comportement du programme, à la fois dans le cas général, mais aussi sur des cas limites (fichier vide, ligne d'exactement 60 caractères, coupures multiples de lignes, dernière ligne sans `'\n'`...).

Pour valider plus facilement vos programmes, vous pouvez dans une version initiale couper les lignes à partir du 6e plutôt que du 60e caractère.

Exercice 28
Validez votre programme avec le jeu d'essai défini. Vous utiliserez pour cela les redirections des entrées/sorties des commandes proposées par le shell.

Cette démarche
– spécification.
– développement.
– test et validation.
doit être systématique.

9 Rendre vos travaux

Dans le cadre de ces enseignements, vous utiliserez (exclusivement) une application nommée PROF, Programme de Remise Officiel de Fichiers, pour rendre vos travaux.

Cette application est accessible dans l'Intranet du FIL depuis les salles TP. Vous ne pouvez y accéder de l'extérieur du réseau du FIL.

Cette application gère une date butoir avant laquelle il est impératif que vous rendiez votre TP.

Vous travaillez normalement en binôme. Vous allez rendre un unique TP pour votre binôme et vous-même.

Rendre en TP consiste à créer une archive contenant vos fichiers et à déposer cette archive sur PROF. Une archive est un fichier qui « contient » un ensemble de fichiers. Vous utiliserez la commande `tar` pour créer une archive selon le modèle suivant :

```
% tar czf archive.tgz file1.c file2.c Makefile Readme
```

Consultez au besoin le manuel de `tar`.

Il est entendu qu'il n'est pas utile (et mal vu) de rendre les fichiers objets (`.o`) ou les exécutables : ceux-ci seront reconstruits grâce à `make` à partir des fichiers sources que vous allez rendre.

Vous pouvez rendre de multiples versions de vos TP. N'attendez donc pas le dernier moment pour rendre vos TP.

Pour plus de précisions, consultez la documentation de PROF disponible sur l'intranet du FIL à `http://intranet.fil.univ-lille1.fr/INTRANET/PROF/Doc.html`.

Exercice 29

Ce TP est aussi un TP d'initiation à l'utilisation de PROF... Vous devez rendre via PROF les fichiers suivants :

- `fbib.c`, `fbib.h`, `fact.c`;
- `tronque.c`;
- un répertoire `jeuxsctr` contenant les jeux d'essai utilisés pour valider votre commande `tronque` (exercice 27) ;
- `Makefile`
- `Readme`, un fichier donnant toute information utile sur vos travaux :
  - votre nom, celui de votre binôme,
  - la définition d'un mot (exercice 25),
  - etc.

10 Recommandation finale

Nous vous demandons (et exigeons pour les projets) de toujours utiliser les options de compilation `-Wall` (signaler tous les warnings), `-Werror` (considérer les warnings comme des erreurs), `-ansi` (le source est du C-ANSI), et `-pedantic` (et seulement du C-ANSI).

Nous exigeons également que vous rendiez vos projets avec un `Makefile`, le correcteur ne doit avoir qu'à exécuter `make` !