Programmation C

Fiche technique n° 2 : la commande `make`

À quoi servent la commande `make` et les fichiers `Makefile` ?

Premier problème : des commandes de compilation fastidieuses

Pour un projet C (ou autre) contenant plusieurs fichiers répartis dans plusieurs répertoires et utilisant plusieurs options de compilation, compiler devient fastidieux. Par exemple, imaginons un TP où :

- Les fichiers sources `.c` sont dans un sous-répertoire `src`.
- Les fichiers `.h` sont dans un sous-répertoire `include`.
- On veut produire les fichiers objets `.o` dans un sous-répertoire `build`.

On pourrait compiler chaque fichier `.c` depuis le répertoire principal par une commande comme :

```
$ gcc -c -I./include -std=gnu23 -Wall -pedantic -Wextra \
      -o build/matrix1.o src/matrix1.c
```

Dans cet exemple, le caractère `\` (*backslash*) en fin de première ligne sert à indiquer au shell que la commande n'est pas terminée : elle continue sur la ligne suivante.

On rappelle (voir diaporama ou polycopié) que l'option `-c` demande au compilateur `gcc` de ne pas créer l'exécutable, juste le fichier `build/matrix1.o` à partir du fichier `src/matrix1.c`. L'option `-I./include` indique à `gcc` le répertoire `include` pour y chercher des fichiers d'en-tête `.h`.

Cette commande est fastidieuse à taper, et en réalité, on compilera avec davantage d'options. Voici par exemple à quoi ressemblera la commande de compilation qu'on utilisera pour le TP 1 :

```
$ gcc -c -I./include -std=gnu23 -MMD -Wall -pedantic          \
      -Wextra -Wpointer-arith -Wstrict-prototypes             \
      -Wmissing-prototypes -Wuninitialized                    \
      -Wimplicit-function-declaration -Wunused-result         \
      -Wshadow -Wformat=2 -Wnull-dereference -Wvla            \
      -Wwrite-strings -fsanitize=address,undefined            \
      -fno-sanitize-recover=undefined                         \
      -fno-omit-frame-pointer -Wcast-qual -g -O0 -Werror      \
      -Wno-error=unused-parameter -Wno-error=unused-function \
      -o build/matrix1.o src/matrix1.c
```

On apprécie peu d'avoir à taper une telle commande : ça n'a aucun intérêt, et même en utilisant le copié-collé, on risque de faire des erreurs. De plus, il faut l'adapter pour compiler chaque fichier `.c`. Pour créer l'exécutable, il faudra enfin lancer une dernière compilation pour lier entre eux (avec le *linker*) tous les fichiers `.o` qui ont été produits dans le répertoire `build`.

Second problème : des dépendances difficiles à mémoriser

Un autre point important est que, lorsqu'on a plusieurs fichiers sources (.c), il n'est utile de recompiler que ceux qui ont changé ou qui dépendent d'un fichier qui a lui-même changé. Prenons l'exemple d'un projet qui a 30 fichiers sources. Supposons que :

- Depuis la dernière compilation, seulement un fichier source, `util.c`, a été modifié.
- Depuis la dernière compilation, aucun fichier d'en-tête (`.h`) n'a été modifié.

Dans ce cas, il n'est pas utile de recompiler les fichiers autres que `util.c`, car on dispose déjà de leur `.o` associé grâce à la compilation précédente. Il suffit de compiler `util.c`, puis de recréer l'exécutable.

Mais un problème apparaît avec une situation légèrement différente. Supposons maintenant que :

- Depuis la dernière compilation, aucun fichier source (`.c`) n'a été modifié.
- Depuis la dernière compilation, seulement un fichier d'en-tête, `util.h`, a été modifié.

Il est possible que le fichier qui a changé, `util.h`, contienne la définition d'une constante (comme `#define MAX 1024`) utilisée dans les fichiers qui l'incluent. Il faut donc recompiler tous les fichiers `.c` qui incluent `util.h`. Cette inclusion peut être directe, par une directive `#include "util.h"`. Mais elle peut être aussi indirecte : on peut inclure un autre fichier `.h`, qui lui-même inclut `util.h` (à nouveau, soit directement par `#include "util.h"`, soit indirectement). On dit que `util.h` est une *dépendance* de ces fichiers. Retrouver les dépendances entre fichiers pour savoir lesquels compiler est très fastidieux.

Le programme `make`

Le programme `make` est conçu pour gérer les deux problèmes précédents :

- Il détermine automatiquement quels fichiers ont besoin d'être recompilés.
- Il compile ces fichiers, et uniquement ceux-ci, en utilisant les options voulues.

Pour utiliser `make`, on écrit un fichier `Makefile` décrivant les fichiers à compiler, les dépendances entre fichiers et les options de compilation. Heureusement, les dépendances peuvent être calculées automatiquement par le compilateur, ce qui nous dispense de le faire. C'est à cela que sert l'option `-MMD` que vous avez peut-être remarquée dans la commande de compilation donnée plus haut. Une fois les dépendances entre fichiers déterminées, `make` inspecte les dates de modification de chaque fichier qu'il doit potentiellement recompiler. Un fichier `.c` est recompilé si son `.o` n'existe pas encore, ou si l'un des fichiers dont dépend ce `.c` est plus récent que lui. Ainsi, le programme `make` ne lance que les compilations nécessaires. Un fichier `Makefile` vous sera toujours fourni. **Ne le modifiez pas !**

> L'objectif de cette feuille n'est pas d'apprendre à écrire un `Makefile`.
> C'est de comprendre ce que fait le programme `make` et de savoir l'utiliser.

Voici comment vous pourrez utiliser `make` avec le `Makefile` fourni dans les TP. On peut donner à `make` des options, puis des cibles. Les cibles indiquent ce qu'on veut reconstruire. Cela peut être, par exemple, un fichier `.o`. En pratique, on utilisera les cibles suivantes dans les TP :

- `make <nom_executable>` construit l'exécutable (s'il n'est pas déjà à jour) à partir de tous les fichiers du répertoire `src`. C'est aussi le comportement de la commande `make` sans nom de cible.
- `make compile_commands.json` crée un fichier `compile_commands.json` qui sera lu par votre éditeur (VS Code) pour savoir par exemple comment accéder aux fichiers d'en-tête `.h`.
- `make clean` nettoie le projet (supprime les `.o`, l'exécutable, les fichiers de dépendance `.d`, etc.). Contrairement aux autres cibles citées, `clean` ne désigne pas un nom de fichier à construire, ce qui explique qu'il s'exécute toujours, indépendamment de l'état des fichiers.

Voici quelques options utiles, qu'on peut combiner (voir `man make` ou `info make` pour des détails).

- `-k` : continue la compilation autant que possible, sans s'arrêter dès la première erreur.
- `-B` : force `make` à recompiler tout le projet, comme si aucun fichier n'avait encore été généré.
- `-n` : simule l'exécution en affichant les commandes qui seraient lancées, sans les exécuter.