Programmation en langage C - Cahier 1

Licence informatique 2ème année

Université de La Rochelle

<christophe.demko@univ-lr.fr>

2020-2021_1

Ce cahier de travaux pratiques constitue à notre sens la base des exercices à savoir réaliser pour une pratique basique de la programmation en langage C.

Les lignes commençant par `$` représente une commande *unix*. Le `$` ne fait pas partie de la commande proprement dite mais symbolise l'invite de commande.

Table des matières

Table des figures

Liste des exercices

1 Prise en main de l'environnement

1.1 Utilisation de l'utilitaire make

Sous *unix* (et donc sous *linux* par extension), il existe un utilitaire appelé make (prononcer *mec*), permettant de gérer les processus de création de fichiers. C'est un outil indispensable en programmation puisqu'il permet de maintenir une cohérence temporelle entre les fichiers sources et les fichiers compilés ou exécutables.

make est préfiguré avec des dizaines de règles de création, et notamment pour ce qui nous intéresse, avec une règle de création d'un exécutable à partir d'un fichier .c.

Il suffit pour cela d'exécuter la commande

```
$ make <executable>
```

et l'utilitaire exécutera une ligne de compilation et d'édition des liens (nous reverrons plus tard ces notions distinctes) permettant de créer executable à partir d'executable.c

Exercice 1 *(Création d'un exécutable minimal)*
Créez un fichier exo01.c contenant le nombre minimum d'instructions pour créer un exécutable. Lancez la commande permettant de créer l'exécutable exo01.

Exercice 2 *(Étude des règles par défaut de l'utilitaire make)*
En utilisant l'option -p de l'utilitaire make, générez un fichier Makefile dans le dossier temporaire /tmp avec la commande suivante :

```
$ make -p > /tmp/Makefile
```

contenant toutes les règles par défaut de l'outil make (*vous utiliserez à bon escient le manuel de*

make). Éditez ce fichier avec votre éditeur favori (`gedit`). *La création de ce fichier n'est là que pour étudier les règles par défaut ; il ne s'agit pas d'un fichier pérenne*.

Les variables intéressantes à configurer pour la programmation C sont au nombre de 6 :

CPPFLAGS options du préprocesseur CPreProcessorFLAGS).

CFLAGS options du langage C.

LDFLAGS options de l'aggrégateur de liens (*linker* en anglais). À l'origine, *ld* était un acronyme pour *load*).

LDLIBS options de l'aggrégateur de liens ne contenant que les librairies à lier (LoaDLIBrarieS)

LOADLIBES options de l'aggrégateur de liens ne contenant que les répertoires des librairies à lier

TARGET_ARCH le processeur cible de la compilation ; il est en effet possible de compiler sous *unix* pour une exécution prévue sous *windows* ou sous *macos*

Exercice 3 *(Variables pour la compilation C)*

1. Quelles sont les règles permettant de créer un exécutable à partir d'un fichier écrit en C ? Comment sont utilisées les variables précédemment citées ?
2. Quelles sont les règles permettant de créer un fichier objet à partir d'un fichier écrit en C? Comment sont utilisées les variables précédemment citées ?

1.2 Utilisation du débogueur gdb

1.2.1 Configuration

Un débogueur, ou *dévermineur* pour être précis, est un outil permettant d'observer le déroulement d'un programme. C'est l'*outil* indispensable du *bon* programmeur, en opposition avec celui qui préfère ajouter à son code des impressions de valeur intermédiaires pour les effacer lorsque le programme est finalisé.

Pour utiliser le débogueur (gdb dans le cas présent), il suffit de compiler et d'éditer les liens avec l'option -g. `make` a prévu un mécanisme pour le faire naturellement. Si un fichier nommé `Makefile` ou `makefile` est présent dans le répertoire courant, `make` s'en sert comme entrée pour les règles de création. Pour ce qui nous concerne, il suffit donc de créer un `Makefile` contenant deux lignes

```
CFLAGS=-g
LDFLAGS=-g
```

`CFLAGS=-g` indique au compilateur l'ensemble des options (`CFLAGS`) qu'il devra prendre en compte (ici -g).

`LDFLAGS=-g` indique à l'éditeur des liens l'ensemble des options (`LDFLAGS`) qu'il devra prendre en compte (ici -g également). Chaque exécutable sera alors créé pour pouvoir être exécuté avec le débogueur. Il suffit pour cela d'exécuter la commande

```
$ gdb <executable>
```


1.2.2 Utilisation

Plusieurs commandes permettent d'utiliser gdb. Tout au long des TP, nous les verrons en détail.

Voici les principales à connaître :

`help` menu général de l'aide;

`list` permet de visualiser le code source du programme;

`run` permet de lancer l'exécution du programmre;

`jump numéro` permet de reprendre l'exécution du programme à la ligne `numéro`;

`until numéro` permet de continuer le programme jusqu'à la ligne `numéro`;

`step` ou `next` permet d'exécuter la ligne suivante (nous verrons par la suite la différence entre ces deux commandes);

`finish` termine l'exécution;

`print exp` permet de connaître la valeur de l'expression `exp`;

`set var variable_name=exp` permet de modifier la valeur de la variable `variable_name` et de lui assigner la valeur `exp`;

`display exp` permet d'imprimer la valeur de l'expression `exp` à chaque fois que le programme s'arrête;

`undisplay num` retire de la liste des expressions à afficher l'expression de numéro num;

`x &var` permet d'examiner la mémoire de la variable `var` avec différent formats

`break numéro` permet de faire arrêter l'exécution du programme à la ligne `numéro`;

`delete numéro` permet de d'effacer le *break* `numéro` (attention, ce n'est pas le numéro de ligne mais le numéro du *break*).

Il faut noter que la plupart de ces commandes sont utilisables avec des abbréviations (`l` pour `list`, etc).

Exercice 4 *(Utilisation du débogueur)*

1. Écrire le programme suivant dans un fichier `gcd.c`

```c
#include <stdlib.h>

int main(void) {
  int a, b;
  a %= b;
  b %= a;
  return EXIT_SUCCESS;
}
```

2. Compiler le programme pour pouvoir l'utiliser avec le débogueur;
3. Lancer le programme avec le débogueur;
4. Affecter à la variable a la valeur 32 et à la variable b la valeur 56;

5. À l'aide des commandes du dégogueur, exécuter la ligne `a %= b;`
   — si a est plus grand que b
   et la ligne `b %= a;`
   — si b est plus grand que a jusque l'un d'entre eux soit nul.

Exercice 5 *(Compilation séparée)*
La fonction *Plus Grand Commun Diviseur* (*GreatestCommonDivisor*) peut être intéressante à définir dans un fichier séparé de la fonction principale `main`. Pour permettre une extension aisée des fonctions arithmétiques, nous l'écrirons dans un fichier `arithmetic.c` et nous la nommerons `arithmetic_gcd`.

1. Écrire dans un fichier `arithmetic.c` une fonction `arithmetic_gcd` permettant de calculer le *pgcd* de deux nombres. Le code sera contenu dans une fonction dont la déclaration est :

   ```
   extern unsigned int arithmetic_gcd(unsigned int a,unsigned int b);
   ```

2. Écrire un fichier `gcd-32-56.c` permettant d'afficher le *pgcd* de 32 et 56 :

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

   extern unsigned int arithmetic_gcd(unsigned int a, unsigned int b);

   int main(void) {
     int a = 32;
     int b = 56;
     printf("%u\n", arithmetic_gcd(a, b));
     return EXIT_SUCCESS;
   }
   ```

3. Écrire dans le `Makefile` les instructions pour créer l'exécutable `gcd-32-56` à partir de `gcd-32-56.o` et `arithmetic.o` :

   ```
   CFLAGS=-g
   LDFLAGS=-g

   all: gcd-32-56

   gcd-32-56: gcd-32-56.o arithmetic.o
   ```

   Testez le résultat en exécutant directement

   ```
   $ make
   ```

   La cible `all` est exécuté lorsqu'aucun argument n'est passé à la commande `make`. La cible `all` dépend de `gcd-32-56` qui est donc créé après création des `.o` qui dépendent par défaut de leurs `.c` respectifs.

4. Écrire un fichier `arithmetic.h` permettant de remplacer la ligne

   ```
   extern unsigned int arithmetic_gcd(unsigned int a, unsigned int b);
   ```

   par une directive d'inclusion du préprocesseur. Testez le résultat.

1.3 Utilisation de l'utilitaire `cmake`

cmake [1] est un utilitaire multi-plateforme permettant, à l'aide d'un langage de définition de projets, de définir comment et quels sont les fichiers à créer dans un projet de programmation. Il fonctionne aussi bien sous *window*, *unix*, ou *macos* et permet de générer des projets prêts à être utilisés avec :

- l'utilitaire `make`
- l'environnement de développement pour *windows* *Visual Studio* [2]
- l'environnement de développement pour *windows*, *unix*, ou *macos* *CodeBlocks* [3]
- l'environnement de développement pour *windows*, *unix*, ou *macos* *CodeLite* [4]
- l'environnement de développement pour *windows*, *unix*, ou *macos* *Eclipse* [5]
- l'environnement de développement pour *windows*, *unix*, ou *macos* *Sublime Text* [6]
- l'environnement de développement pour *unix* ou *macos* *KDevelop* [7]
- l'environnement de développement pour *unix* ou *macos* *Kate* [8]
- l'environnement de développement pour *macos* *XCode* [9]

La définition du projet se fait dans un fichier nommé `CMakeLists.txt`.

Exercice 6 *(`CMakeLists.txt` minimal)*
Créez un dossier `src/` et copiez y les fichiers `gcd-32-56.c`, `arithmetic.c` et `arithmetic.h` précédemment créés. Avec votre éditeur de texte favori, créez dans ce dossier `src/` un fichier `CMakeLists.txt` contenant :

```
cmake_minimum_required(VERSION 3.0)
project(Arithmetic C)
add_executable(gcd-32-56 gcd-32-56.c arithmetic.c arithmetic.h)
```

La ligne `add_executable(gcd-32-56 gcd-32-56.c arithmetic.c arithmetic.h)` indique à `cmake` qu'il faudra créer un exécutable nommé `gcd-32-56` à partir des fichiers sources `gcd-32-56.c` et `arithmetic.c` (`arithmetic.h` n'est pas utilisé pour l'aggrégation : par convention, `cmake` n'utilise que les dates de modification des fichiers d'entête pour déterminer si l'aggrégation doit être relancé). `cmake` reconnaît le type des fichiers d'après leur extension : `gcd-32-56.c` et `arithmetic.c` seront compilés pour créer leurs `.o` respectifs et aggrégés pour créer l'exécutable `gcd-32-56`.
Créez un dossier `build/` au même niveau que `src/` :

```
$ mkdir build
$ tree
.
```

1. https://cmake.org/
2. https://www.visualstudio.com/
3. http://www.codeblocks.org/
4. http://codelite.org/
5. https://eclipse.org/
6. http://www.sublimetext.com/
7. https://www.kdevelop.org/
8. http://kate-editor.org/
9. https://developer.apple.com/xcode/


```
+-- build
+-- src
    +-- arithmetic.c
    +-- arithmetic.h
    +-- CMakeLists.txt
    +-- gcd-32-56.c

2 directories, 4 files
```

Placez vous dans le dossier `build/` et exécutez la ligne de commande `cmake ../src/` :

```
$ cd build/
$ cmake ../src/
-- The C compiler identification is GNU 5.2.1
-- The CXX compiler identification is GNU 5.2.1
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: .../build
```

À ce stade, le dossier `build/` est prêt pour la compilation et l'édition des liens. Tout un aréopage de fichiers ont été créés par l'utilitaire `cmake`.

```
$ tree
.
+-- CMakeCache.txt
+-- CMakeFiles
|   +-- 3.2.2
|   |   +-- CMakeCCompiler.cmake
|   |   +-- CMakeCXXCompiler.cmake
|   |   +-- CMakeDetermineCompilerABI_C.bin
|   |   +-- CMakeDetermineCompilerABI_CXX.bin
|   |   +-- CMakeSystem.cmake
|   |   +-- CompilerIdC
|   |   |   +-- a.out
|   |   |   +-- CMakeCCompilerId.c
```

```
|   |   +-- CompilerIdCXX
|   |       +-- a.out
|   |       +-- CMakeCXXCompilerId.cpp
|   +-- cmake.check_cache
|   +-- CMakeDirectoryInformation.cmake
|   +-- CMakeOutput.log
|   +-- CMakeTmp
|   +-- feature_tests.bin
|   +-- feature_tests.c
|   +-- feature_tests.cxx
|   +-- Makefile2
|   +-- Makefile.cmake
|   +-- gcd-32-56.dir
|   |   +-- build.make
|   |   +-- cmake_clean.cmake
|   |   +-- DependInfo.cmake
|   |   +-- depend.make
|   |   +-- flags.make
|   |   +-- link.txt
|   |   +-- progress.make
|   +-- progress.marks
|   +-- TargetDirectories.txt
+-- cmake_install.cmake
+-- Makefile

6 directories, 29 files
```

Il n'y a plus qu'à exécuter `make` dans le dossier `build/` :

```
$ make
Scanning dependencies of target gcd-32-56
[ 50%] Building C object CMakeFiles/gcd-32-56.dir/gcd-32-56.c.o
[100%] Building C object CMakeFiles/gcd-32-56.dir/arithmetic.c.o
Linking C executable gcd-32-56
[100%] Built target gcd-32-56
```

L'exécutable `gcd-32-56` a été créé ; il n'y a plus qu'à l'exécuter.

```
$ ./gcd-32-56
8
```

Il est possible de demander à la commande `make` d'afficher les commandes exécutées en ajoutant `VERBOSE=1` :

```
$ make VERBOSE=1
```

La commande `make` accepte des arguments supplémentaires. Pour les connaître :

```
$ make help
```

Exercice 7 *(Création de bibliothèque)*
Pour créer une bibliothèque, il faut indiquer à cmake les fichiers sources dont elle dépend :

```
cmake_minimum_required(VERSION 3.0)

project(Arithmetic C)

add_executable(gcd-32-56 gcd-32-56.c arithmetic.c arithmetic.h)
add_library(arithmetic SHARED arithmetic.c arithmetic.h)
```

La ligne `add_library(arithmetic SHARED arithmetic.c arithmetic.h)` indique à cmake qu'il faudra créer une librairie partagée nommée `arithmetic` à partir du fichier `arithmetic.c`.
Relancez la compilation et vérifiez que la librairie est bien créée dans le dossier `build/`.

Exercice 8 *(Installation de fichiers)*
Avoir créer une bibliothèque ne suffit pas, l'idéal est de l'installer dans un dossier où elle pourra être utilisée à loisir. Une bibliothèque s'accompagne généralement d'un fichier d'entête définissant le contrat entre le créateur de la bibliothèque et son utilisateur. Pour installer des fichiers avec cmake, ajouter les lignes suivantes dans le fichier `CMakeLists.txt` :

```
cmake_minimum_required(VERSION 3.0)

project(Arithmetic C)

add_executable(gcd-32-56 gcd-32-56.c arithmetic.c arithmetic.h)
add_library(arithmetic SHARED arithmetic.c arithmetic.h)

install(
    TARGETS arithmetic
    LIBRARY DESTINATION lib
    ARCHIVE DESTINATION lib
    RUNTIME DESTINATION bin
)

install(
    FILES arithmetic.h
    DESTINATION include
)
```

Les lignes `install` indique à cmake où installer les fichiers. Les librairies et les archives sont installées dans un dossier `lib/`, les exécutables dans un dossier `bin/` et l'entête dans un dossier `include/`. Ces dossiers sont considérés relativement à la variable de cmake `CMAKE_INSTALL_PREFIX` qui peut se définir soit lorsque l'on lance cmake sur la ligne de commande, soit par l'intermédiaire de l'exécutable `cmake-gui`. Il est à noter que sous *windows*, les

librairies partagées sont considérées comme des exécutables.

```
$ cmake ../src -DCMAKE_INSTALL_PREFIX=/tmp
```

Le préfixe d'installation `/tmp` est donné ici à titre d'exemple. Généralement, sous environnement *unix*, l'utilisateur préférera utiliser le dossier `.local` de son compte (`~/.local`).
Si l'on a pas défini la variable `CMAKE_INSTALL_PREFIX` lors du lancement de `cmake`, 2 options sont possibles :

- lancer
  `$ make edit_cache`
  et modifier la valeur de la variable `CMAKE_INSTALL_PREFIX` avec l'interface proposée
- lancer (sous système *Unix* seulement)
  `$ make DESTDIR=/path/to/install install`
  pour installer le projet dans le chemin `/path/to/install`

Après avoir modifié la valeur de la variable `CMAKE_INSTALL_PREFIX`, lancez la compilation et procédez à l'installation de la librairie en lançant la commande suivante :

```
$ make install
```

Les fichiers devront être copiés dans les dossiers décrits.
Il est possible d'indiquer à `cmake` la configuration demandée à l'aide de la définition de la variable `CMAKE_BUILD_TYPE` :

- `Debug` mode déboggage
- `Release` mode final
- `RelWithDebInfo` mode final avec informations de déboggage
- `MinSizeRel` mode final optimisé

```
$ cd ..
$ mkdir debug
$ cd debug
$ cmake ../src -DCMAKE_INSTALL_PREFIX=/tmp -DCMAKE_BUILD_TYPE=Debug
$ make
$ make install
```

Exercice 9 *(Ajout de tests)*
Une pratique extrèmement répandue en programmation consiste à définir des tests unitaires en même temps que sont définies les fonctions d'une bibliothèque. `cmake` permet de gérer les tests unitaires. Pour utiliser les tests unitaires avec `cmake`, ajouter les lignes suivantes dans le fichier `CMakeLists.txt` :

```
cmake_minimum_required(VERSION 3.0)

project(Arithmetic C)

add_executable(gcd-32-56 gcd-32-56.c arithmetic.c arithmetic.h)
add_library(arithmetic SHARED arithmetic.c arithmetic.h)
```

```
install(
    TARGETS arithmetic
    LIBRARY DESTINATION lib
    ARCHIVE DESTINATION lib
    RUNTIME DESTINATION bin
)

install(
    FILES arithmetic.h
    DESTINATION include
)

add_executable(test-gcd test-gcd.c)
add_dependencies(test-gcd arithmetic)
target_link_libraries(test-gcd arithmetic)

enable_testing()
add_test(test-gcd ./test-gcd)
```

Les lignes

```
add_executable(test-gcd test-gcd.c)
add_dependencies(test-gcd arithmetic)
target_link_libraries(test-gcd arithmetic)
```

permettent de

- définir un nouvel exécutable dépendant du fichier source `test-gcd.c`
- préciser que cet exécutable doit être créé après la librairie `arithmetic`
- préciser que cet exécutable utilisera la librairie `arithmetic`

Les lignes

```
enable_testing()
add_test(test-gcd ./test-gcd)
```

permettent de

- utiliser le système de tests unitaires de `cmake`
- ajouter l'exécutable `./test-gcd` comme étant un test à exécuter

Le fichier `test-gcd.c` devra retourner 0 (succès) si l'ensemble des tests s'est bien déroulés. Par exemple :

```
#include <stdlib.h>

#ifdef NDEBUG
#undef NDEBUG
#endif
#include <assert.h>
```

```c
#include "./arithmetic.h"

int main(void) {
  assert(arithmetic_gcd(32, 56) == 8);
  assert(arithmetic_gcd(101, 23) == 1);
  assert(arithmetic_gcd(101, 0) == 101);
  assert(arithmetic_gcd(0, 101) == 101);
  assert(arithmetic_gcd(0, 0) == 0);
  return EXIT_SUCCESS;
}
```

La macro `assert` permet de vérifier une condition et de sortir du programme avec un code d'erreur le cas échéant. Elle n'est active que si la constante NDEBUG n'est pas définie.
Après la compilation, il est possible de lancer les tests unitaires :

```
$ make test
Running tests...
Test project .../build
    Start 1: test-gcd
1/1 Test #1: test-gcd .........................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.00 sec
```

1.4 Utilisation d'un IDE

Si l'on veut générer d'autres types de projet que celui généré par défaut (`make` en ligne de commande), il suffit de lancer `cmake` avec un paramètre supplémentaire :

```
$ cmake ../src -G <generator-name>
```

Pour connaître la liste des générateurs disponibles :

```
$ cmake --help
```

Il est également possible d'utiliser l'utilitaire `cmake-gui` fonctionnant sous *unix* et sous *windows*. Cet utilitaire permet de spécifier facilement des variables d'environnement ainsi que le type de projet ciblé. Sur les machines de l'université, on peut facilement utiliser l'IDE `codeblocks` :

— sous *unix*, il faut choisir la version `Codeblocks - Unix Makefiles` lors de la configuration du projet (voir figure 1)

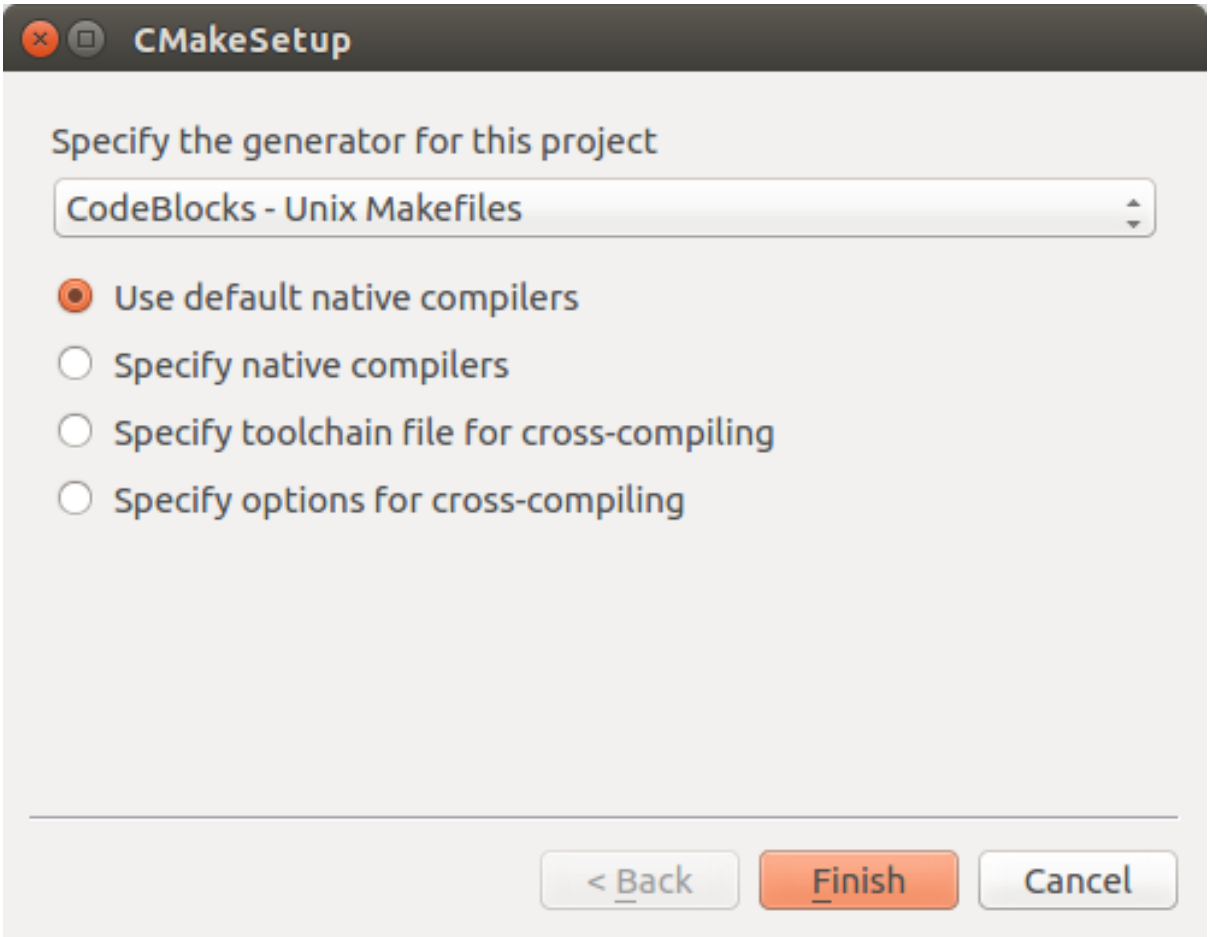

FIGURE 1 – Configuration de `cmake` sous *unix* avec l'interface graphique

— sous *windows*, il faut choisir la version `Codeblocks - Unix Makefiles` lors de la configuration du projet (voir figure 2)

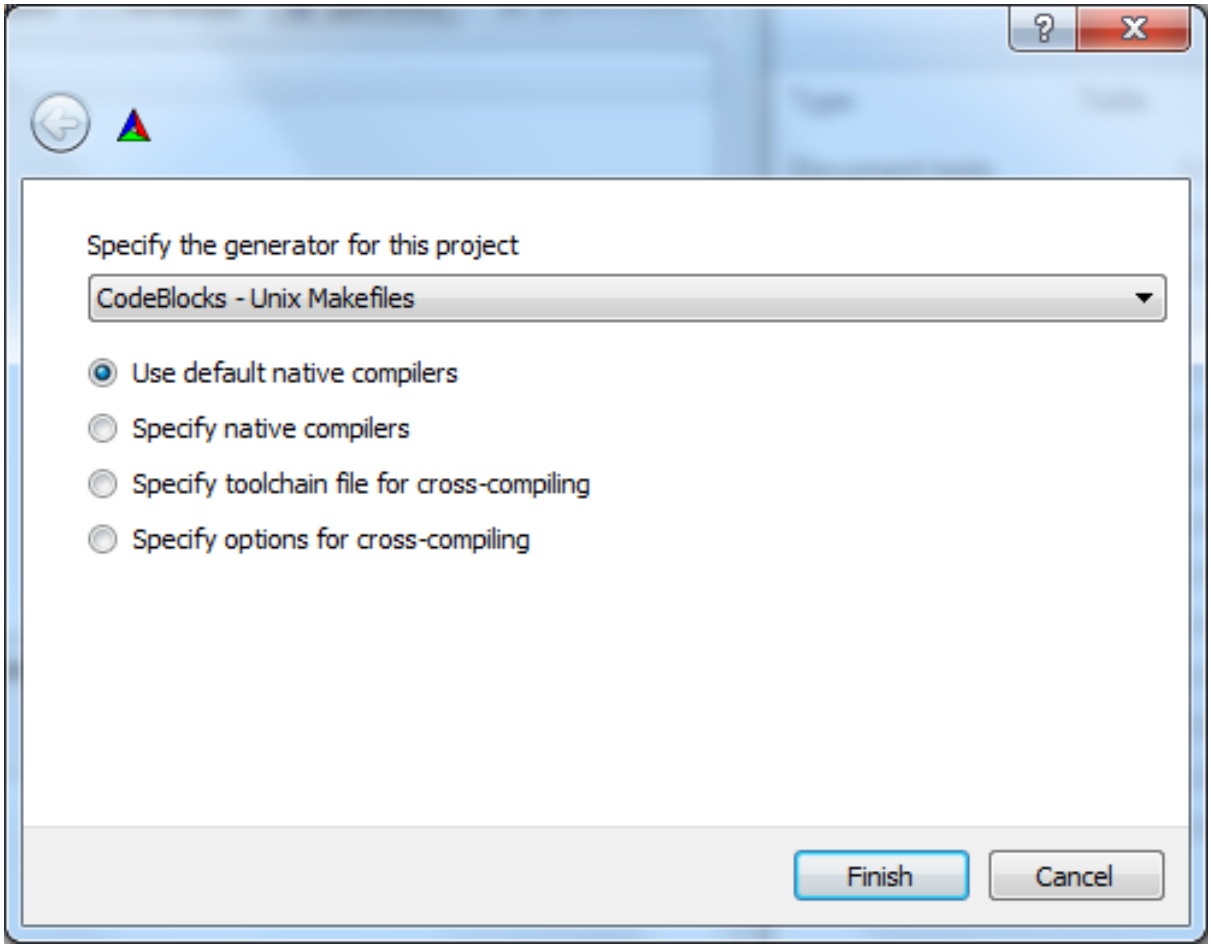

FIGURE 2 – Configuration de cmake sous *windows* avec l'interface graphique

Lorsque le projet a été configuré, il convient de renseigner des variables apparaissant en rouge et éventuellement d'autres variables :

— sous *unix* :

`CMAKE_BUILD_TYPE` Le type de construction (`Debug`, `RelWithDebInfo`, `MinSizeRel` ou `Release`. Voir configuration de `cmake`).

`CMAKE_INSTALL_PREFIX` Le dossier où se fera l'éventuelle installation.

Dans la figure 3, les fichiers sources sont dans `/tmp/src`, le dossier de construction du projet est dans `/tmp/build` et le dossier d'installation est dans le dossier `/tmp/install`.

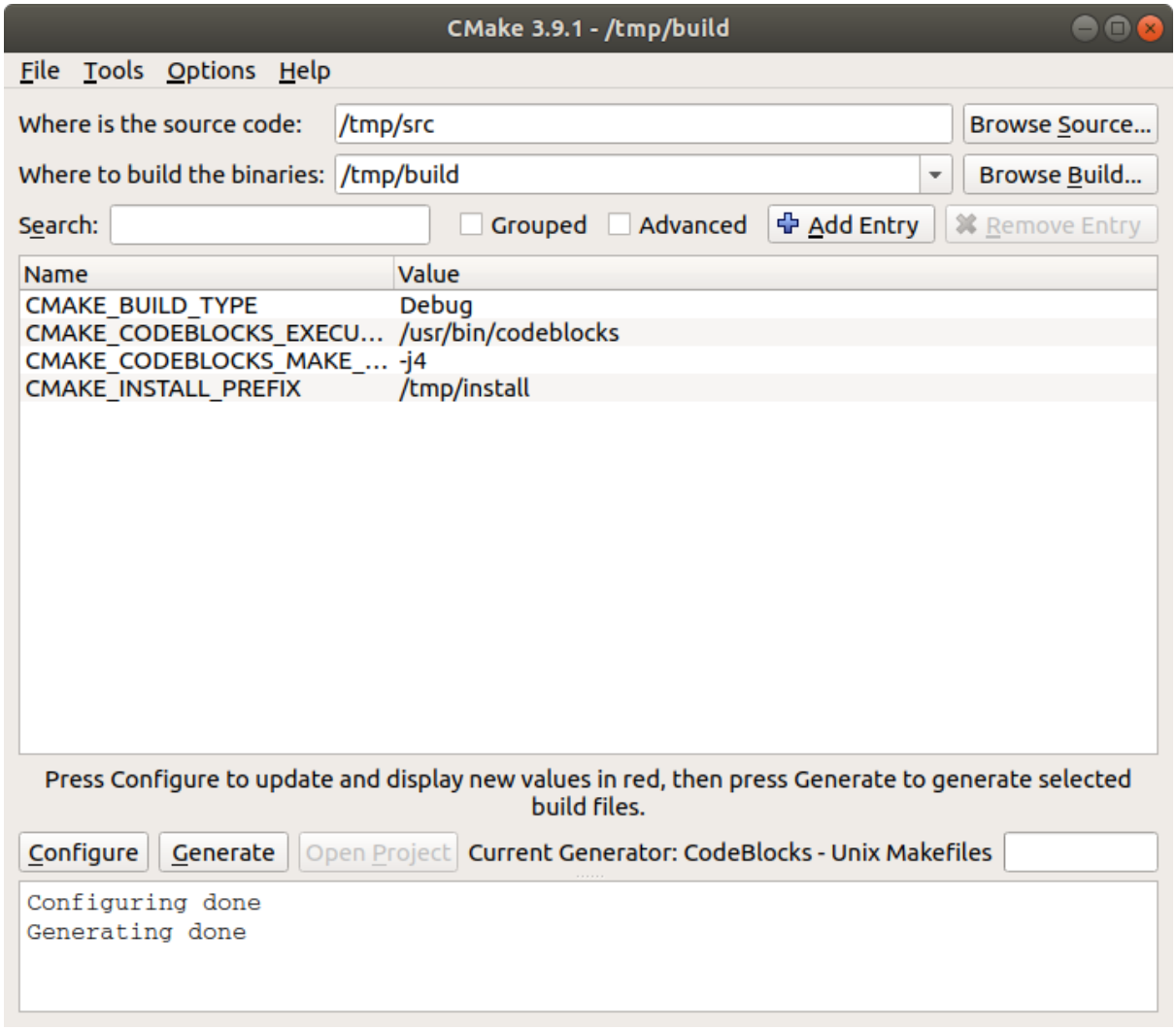

FIGURE 3 – Génération du projet sous *unix*

— sous *windows* :

`CMAKE_BUILD_TYPE` Le type de construction (`Debug`, `RelWithDebInfo`, `MinSizeRel` ou `Release`. Voir configuration de `cmake`).

`CMAKE_CODEBLOCKS_EXECUTABLE` Le chemin pour accéder à l'exécutable `codeblocks`.

`CMAKE_INSTALL_PREFIX` Le dossier où se fera l'éventuelle installation.

Dans la figure 4, les fichiers sources sont dans le dossier `src/` du disque `D`, le dossier de construction du projet est dans le dossier `build/` du disque `D` et le dossier d'installation est dans le dossier `install/` du disque `D`.

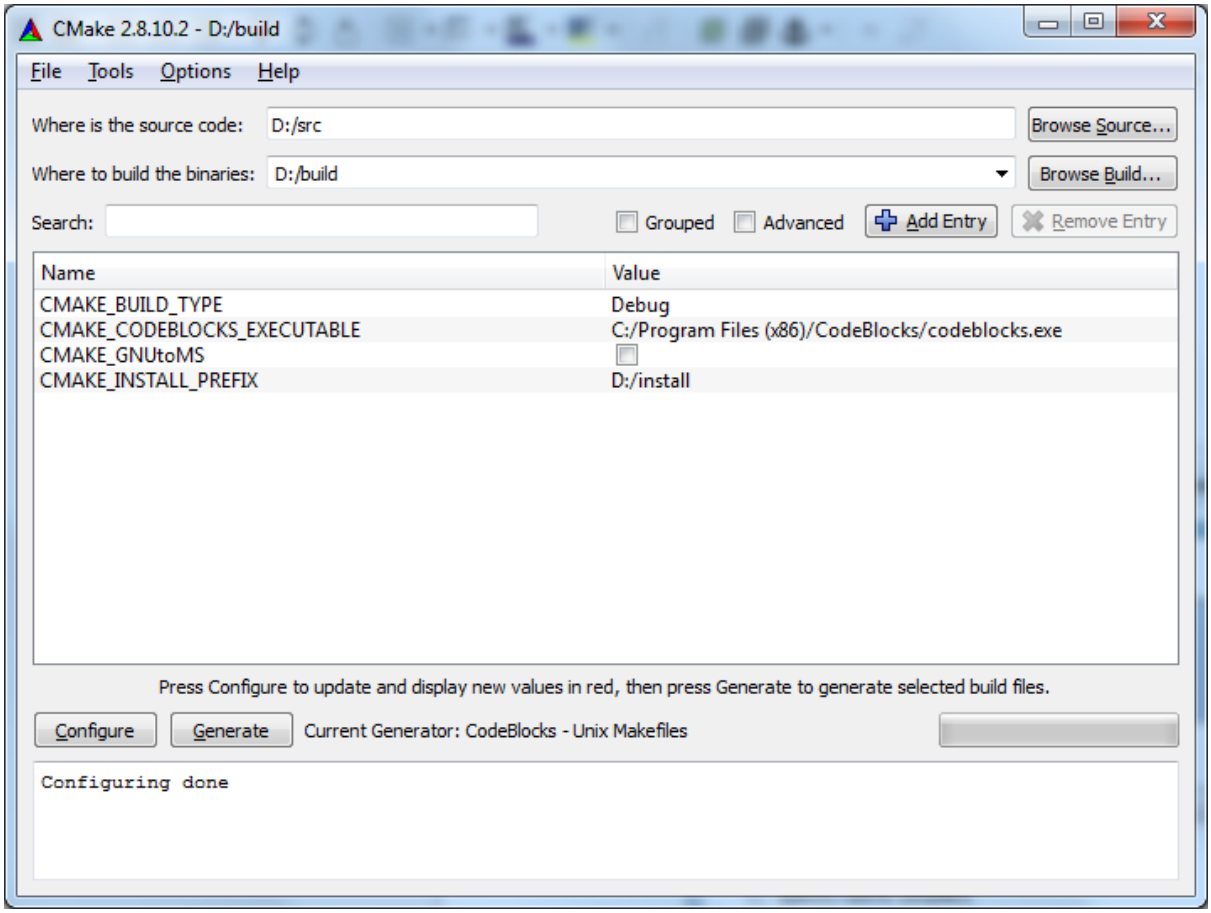

FIGURE 4 – Génération du projet sous *windows*

Exercice 10 *(Utilisation de `codeblocks`)*
L'utilisation de `codeblocks` est assez simple sous les deux systèmes, il suffit d'ouvrir le fichier de projet `codeblocks` généré par `cmake-gui`. Ce fichier porte une extension `cbp` (**C**ode **B**locks **P**roject).
En utilisant les explications fournies, compilez les sources du projet `Arithmetic` en utilisant `codeblocks` et vérifiez que les tests sont toujours fonctionnels.
Étudiez le débermineur fourni par `codeblocks` et exécutez le programme `test-gcd` pas à pas.

2 Manipulation de structures

2.1 Structure

En langage C, les fractions n'existent pas, nous allons créer une bibliothèque de fonctions permettant de manipuler ces structures.

Une fraction peut être représentée par :

— un nombre entier représentant le numérateur
— un nombre entier strictement positif représentant le dénominateur

Une fraction sera toujours stockée réduite (voir fractions [10])

La bibliothèque *fraction* sera créée dans un nouveau projet `cmake`.

Exercice 11 *(Structure de données et entête)*
Dans un fichier `fraction.h`, définir un type de structure `Fraction` permettant de représenter les fractions. Faites en sorte que le contenu du fichier `fraction.h` ne puisse pas être inclus plusieurs fois.

Exercice 12 *(L'opérateur `sizeof`)*
Toutes les données en C ont une taille en octets, en utilisant l'opérateur `sizeof`, quelle est la taille de la structure `Fraction`? Expliquer la valeur obtenue.

2.2 Opérations

Les opérations d'addition, de symétrique, de soustraction, de multiplication, d'inverse et de division sont définies de la manière suivante :

Soient 2 fractions réduites $\frac{n_a}{d_a}$ et $\frac{n_b}{d_b}$.

Les opérations arithmétiques classiques sont définies par :

$$\begin{aligned} a+b &= \text{reduce}\left(\frac{n_a d_b + n_b d_a}{d_a d_b}\right) \\ -a &= \frac{-n_a}{d_a} \\ a-b &= a+(-b) \\ ab &= \text{reduce}\left(\frac{n_a n_b}{d_a d_b}\right) \\ a^{-1} &= \frac{sgn\,(n_a)\,d_a}{sgn\,(n_a)\,n_a} \\ a/b &= a\left(b^{-1}\right) \end{aligned}$$

où `reduce` représente la version réduite de la fraction et `sgn` représente le signe d'un entier.

Vous aurez besoin de la librairie `arithmetic` vue précédemment en ajoutant dans le fichier `CMakeLists.txt` du projet *fraction* les lignes suivantes :

```
find_path(ARITHMETIC_INCLUDE_DIR arithmetic.h)
include_directories(${ARITHMETIC_INCLUDE_DIR})

add_library(fraction SHARED fraction.c fraction.h)
find_library(ARITHMETIC_LIB arithmetic)
target_link_libraries(fraction ${ARITHMETIC_LIB})
```

Ces lignes permettent :

10. http://fr.wikipedia.org/wiki/Fraction_(mathématiques)

- la recherche du fichier `arithmetic.h` dans les chemins spécifiés dans les variables particulières (`CMAKE_PREFIX_PATH` et `CMAKE_INCLUDE_PATH` qui sont déterminées par défaut par `CMAKE_INSTALL_PREFIX`) voir https://cmake.org/cmake/help/latest/command/find_path.html;
- la recherche de la librairie arithmetic dans les chemins spécifiés dans des variables particulières (`CMAKE_PREFIX_PATH` et `CMAKE_LIBRARY_PATH` qui sont déterminées par défaut par `CMAKE_INSTALL_PREFIX`) voir https://cmake.org/cmake/help/latest/command/find_library.html;
- l'aggrégation avec la librairie arithmetic.

La préparation de la compilation peut maintenant se faire en définissant soit

- `CMAKE_INSTALL_PREFIX`
- `CMAKE_PREFIX_PATH` si vous avez installé la librairie *arithmetic* dans un dossier non standard. Sous *unix*, le dossier `.local` du compte utilisateur est un des dossiers standards.

avec l'option `-D` de `cmake`.

Cela ne suffira pas lors de l'exécution :

- sous *unix*, si vous déplacez ensuite ces librairies, il faudra ajouter dans la variable d'environnement `LD_LIBRARY_PATH` le ou les chemins où se trouvent vos librairies (à définir dans `~/.bashrc` par exemple)
- sous *windows*, il faudra ajouter au système le moyen d'accéder aux fichiers `dll` générés en modifiant la variable d'environnement `Path` (sur les machines de l'université de La Rochelle, la déconnexion effacera malheureusement cette configuration…) (voir figure 5 et 6).

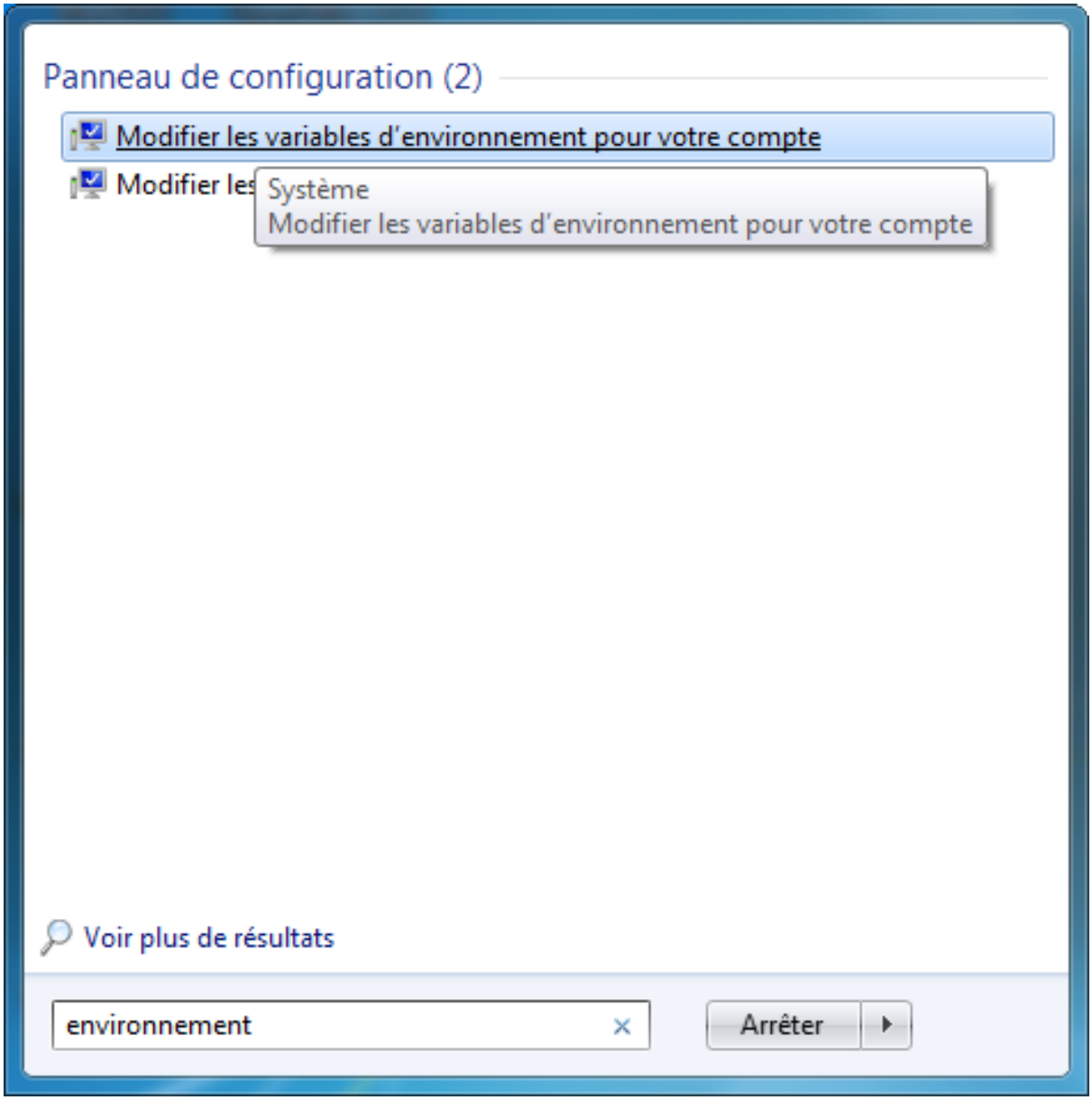

FIGURE 5 – Modifier les variable d'environnement pour votre compte

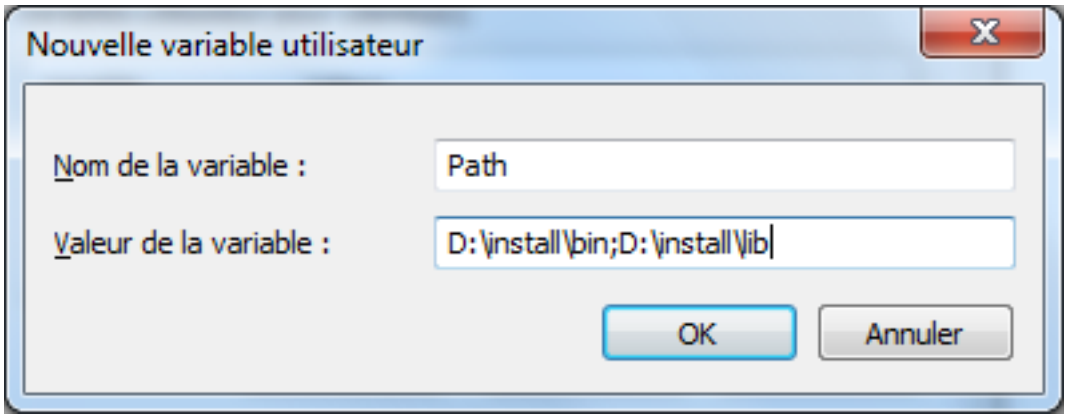

FIGURE 6 – Modifier la variable d'environnement `Path`

Exercice 13 *(Opérations)*

Pour chaque question, écrire un fichier de test (`test-addition.c`, `test-symmetric.c`, ...)

1. Dans le fichier `fraction.c`, écrire une fonction

   ```
   extern Fraction fraction_addition(
     const Fraction a,
     const Fraction b
   );
   ```

   permettant de calculer l'addition de deux fractions.

2. Dans le fichier `fraction.c`, écrire une fonction

   ```
   extern Fraction fraction_symmetric(
     const Fraction a
   );
   ```

   permettant de calculer le symétrique d'une fraction.

3. La soustraction de deux fractions peut être définie comme l'addition de la première avec le symétrique de la seconde. Dans le fichier `fraction.c`, écrire une fonction

   ```
   extern Fraction fraction_substraction(
     const Fraction a,
     const Fraction b
   );
   ```

   permettant de calculer la soustraction de deux fractions.

4. Dans le fichier `fraction.c`, écrire une fonction

   ```
   extern Fraction fraction_multiplication(
     const Fraction a,
     const Fraction b
   );
   ```

   permettant de calculer la multiplication de deux fractions

5. Dans le fichier `fraction.c`, écrire une fonction

   ```
   extern Fraction fraction_inverse(
     const Fraction a
   );
   ```

   permettant de calculer l'inverse d'une fraction.

6. En utilisant le fait que la division est à la multiplication ce que la soustraction est à l'addition, dans le fichier `fraction.c`, écrire une fonction

   ```
   extern Fraction fraction_division(
     const Fraction a,
     const Fraction b
   );
   ```

   permettant de calculer la division de deux fractions.

Exercice 14 *(Initialisation et finalisation)*

Lorsque l'on utilise certaines librairies en programmation, il peut être nécessaire d'appeler une

fonction qui initialise l'utilisation de cette librairie (par exemple: réinitialiser un robot). Tout appel ultérieur à cette fonction d'initialisation n'a aucun effet. Il y a souvent une fonction duale à celle d'initialisation, celle qui finalise l'utilisation de la librairie (par exemple: mettre le robot en mode veille). Le but de cet exercice est de simuler un tel comportement à travers la définition de 2 fonctions :

— `fraction_init` qui simulera l'initialisation de la librairie ;
— `fraction_finish` qui simulera la fin d'utilisation de la librairie.

1. Écrire une fonction

   ```
   extern bool fraction_init(void);
   ```

   permettant d'exécuter un code affichant un message à l'écran seulement la première fois qu'elle est appelée. La fonction renvoie la constante `true` définie dans `<stdbool.h>` si tout s'est bien passé (dans ce cas précis, elle renvoie toujours `true`).

2. Écrire une fonction

   ```
   extern bool fraction_finish(void);
   ```

   permettant d'exécuter un code affichant un message à l'écran seulement la dernière fois qu'elle est appelée. La finalisation (affichage d'un message à l'écran) ne s'exécute que si la fonction `fraction_finish` est appelée le même nombre de fois que la fonction `fraction_init`. La fonction renvoie `true` si tout s'est bien passé (dans ce cas précis, elle renvoie toujours `true`).

3. Écrivez le code suivant dans un fichier `test-init-finish.c` et testez le

   ```
   #include <stdlib.h>

   #ifdef NDEBUG
   #undef NDEBUG
   #endif
   #include <assert.h>

   #include "./fraction.h"

   int main(void) {
     assert(fraction_init());
     assert(fraction_init());
     assert(fraction_init());
     { /* some code */ }
     assert(fraction_finish());
     assert(fraction_finish());
     assert(fraction_finish());
     assert(!fraction_finish());
     return EXIT_SUCCESS;
   }
   ```

   Le code ci-dessus affiche une fois le message d'initialisation (pour son premier appel) et une fois le message de finalisation (pour son troisième appel).

2.3 Utilisation de la mémoire

Il est très rare qu'en programmation, on ait à utiliser directement des structures de données, on préfère dans la plupart des cas utiliser des pointeurs vers ces structures. L'utilisation des pointeurs amène notamment à allouer de la mémoire et à la désallouer.

Exercice 15 *(Remplissage)*
Dans le fichier `fraction.c`, écrire une fonction `fraction_fill_full` de telle manière qu'elle remplisse un pointeur sur fraction existante.

```
extern Fraction *fraction_fill_full(
  Fraction * fraction,
  const int numerator,
  const unsigned int denominator
);
```

Au retour de la fonction, la fraction pointée par `fraction` devra être réduite.
Vous écrirez un fichier de test `test-fill-full.c`

Exercice 16 *(Allocation et désallocation)*
L'écriture d'une bibliothèque gérant une structure de données comprend toujours celle de deux fonctions essentielles : allocation de mémoire et désallocation. Vous écrirez un fichier de test pour chaque question.

1. Dans le fichier `fraction.c`, écrire une fonction

   ```
   extern Fraction *fraction_create_full(
     const int numerator,
     const unsigned int denominator
   );
   ```

   permettant de créer une fraction en mémoire.

   Au retour de la fonction, la fraction devra être retournée réduite.

2. Dans le fichier `fraction.c`, écrire une fonction

   ```
   extern Fraction *fraction_create_default(void);
   ```

   permettant de créer une fraction par défaut. La fraction est créée à partir de 2 variables initialisées à

   — `fraction_numerator_default`
   — `fraction_denominator_default`

   Au retour de la fonction, la fraction devra être retournée réduite.

3. Dans le fichier `fraction.c`, écrire une fonction

   ```
   extern void fraction_destroy(
       Fraction * fraction
   );
   ```

   permettant de libérer la mémoire allouée pour la fraction.

Exercice 17 *(Copie et clonage)*
Vous écrirez un fichier de test pour chaque question.

1. La copie est une opération qui consiste à initialiser une zone mémoire à partir d'une autre zone mémoire. Dans le fichier `fraction.c`, écrire une fonction

   ```
   extern Fraction *fraction_copy(
     Fraction * dest,
     const Fraction * src
   );
   ```

   permettant de copier la fraction `src` vers la fraction `dest` (qui est retournée).

2. Le clonage est une opération qui consiste à allouer de la mémoire et à effectuer une copie à partir d'une autre zone de mémoire. Dans le fichier `fraction.c`, écrire une fonction

   ```
   extern Fraction *fraction_clone(
     const Fraction * src
   );
   ```

   permettant de cloner la fraction `src`.

Exercice 18 *(Retour sur les opérations)*
Modifiez les fonctions de calcul pour qu'elles prennent en compte un pointeur sur fraction plutôt qu'une fraction. Vous modifierez également les fichiers de test.

Exercice 19 *(Transformation en chaînes de caractères)*
Dans le fichier `fraction.c`, écrire une fonction

```
extern const char *fraction_to_string(
  const Fraction * fraction
);
```

permettant de traduire une fraction en chaines de caractères. Cette fonction devra être directement appelable par le code suivant, et ce, sans allocation de mémoire qui pourrait en causer une fuite

```
printf("%s\n", fraction_to_string(fraction));
```

Écrire un fichier de test

Exercice 20 *(Lecture et écriture sur un fichier)*
Vous écrirez un fichier de test pour chaque question.

1. Dans le fichier `fraction.c`, écrire une fonction

   ```
   extern Fraction *fraction_fwrite(
     const Fraction * fraction,
     FILE * stream
   );
   ```

   permettant d'écrire la fraction en *mode binaire* sur le flux `stream`. La fonction renvoie la fraction en cas de succès, `NULL` sinon. Le *mode binaire* signifie que les octets composant la fraction (l'entier signé et l'entier sans signe) sont écrits tels quels. Le fichier doit avoir été ouvert en écriture en dehors de la fonction.

2. Dans le fichier `fraction.c`, écrire une fonction


```
extern Fraction *fraction_fread(
  Fraction * fraction,
  FILE * stream
);
```

permettant de lire la fraction en *mode binaire* sur le flux `stream`. La fonction renvoie la fraction en cas de succès, `NULL` sinon. Cette fonction est la fonction duale de la précédente. Le fichier doit avoir été ouvert en lecture en dehors de la fonction.

2.4 Champs protégés

Une fraction est composée d'un numérateur et d'un dénominateur et est toujours stockée en mémoire réduite. Si la bibliothèque donne accès directement aux champs numérateur et dénominateur, il est possible que l'on se retrouve avec des fractions non réduites. Il faut donc interdire l'accès en écriture aux champs de la fraction.

L'astuce consiste à ne pas définir la structure dans le fichier d'entête `.h` mais à la définir dans le fichier de code `.c`. Ainsi le *contrat* (fichier d'entête) entre le programmeur de la bibliothèque et l'utilisateur de celle-ci est respecté : la modification des champs de la fraction se fera par des *mutateurs* (voir la fonction `fraction_fill_full`) et l'accès par des *accesseurs* (méthodes [11]).

Exercice 21 *(Structure de données et entête)*
Dane le fichier `fraction.h`, supprimer la définition de la structure `_Fraction`. Définir réellement la structure dans un fichier `fraction.inc` qui ne sera inclus que par les fichiers du projet *fraction* mais qui ne sera pas exporté en tant que fichier d'en-tête. Par conséquent, pour les librairies utilisant votre projet *fraction*, il ne devra plus être de possible de créer une fraction non réduite.

Exercice 22 *(*Mutateurs *et* Accesseurs*)*
Vous écrirez un fichier de test pour chaque fonction.
Dans le fichier `fraction.c`, écrire deux fonctions

```
extern int fraction_get_numerator(
  const Fraction * fraction
);
extern unsigned int fraction_get_denominator(
  const Fraction * fraction
);
```

permettant de récupérer le numérateur et le dénominator de la fraction.

2.5 Ne pas utiliser la librairie standard

Certaines fois, pour des besoins de test ou d'utilisation sur des machines n'ayant que peu de mémoire, il peut être utile de créer des exécutables en n'incluant **pas** la librairie standard du C (option `-nostdlib` de l'aggrégateur).

11. https://fr.wikipedia.org/wiki/Méthode_(informatique)

Exercice 23 *(Maîtrise de l'allocation et de la désallocation)*
Ajouter dans le fichier `fraction.h` l'usage de pointeurs de fonction par l'intermédiaire de trois variables globales déclarées par :

```
extern void *(*fraction_malloc)(size_t size);
extern void *(*fraction_realloc)(void *ptr, size_t size);
extern void (*fraction_free)(void *ptr);
```

qui seront définies dans le fichier `fraction.c` par

```
void *(*fraction_malloc)(size_t size) = malloc;
void *(*fraction_realloc)(void *ptr, size_t size) = realloc;
void (*fraction_free)(void *ptr) = free;
```

Les appels à `malloc`, `realloc` et `free` dans le fichier `fraction.c` seront remplacés par des appels à `fraction_malloc`, `fraction_realloc` et `fraction_free`
L'utilisateur de la librairie pourra ensuite modifier ces variables globales pour maîtriser l'allocation.
Écrivez un programme de test `test-my-malloc.c` qui teste cette fonctionnalité.

3 Conclusion et discussions

Ces exercices ne sauraient constituer à eux seuls l'étendue de la programmation en langage C. Je vous encourage à vous exercer le plus souvent possible dans ce langage. Le *Langage C*, c'est :

— 90% d'apprentissage
— et encore 900% de pratique (il n'y a pas d'erreurs de saisie)

Historique des modifications

2012-2013_1 *Dimanche 13 janvier 2013*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Migration de LaTeX vers publican

2012-2013_2 *Lundi 21 janvier 2013*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Correction d'erreurs et ajouts d'exercices

2012-2013_3 *Lundi 21 janvier 2013*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Transfert final du format LaTeX vers publican

2012-2013_4 *Mardi 22 janvier 2013*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Correction de coquilles

2012-2013_5 *Mardi 29 janvier 2013*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Correction de coquilles

2012-2013_6 *Lundi 11 février 2013*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Correction de coquilles

2012-2013_7 *Mardi 12 février 2013*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Correction de coquilles

2012-2013_8 *Samedi 16 mars 2013*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Changement de titre et sous-titre

2013-2014_1 *Vendredi 17 janvier 2014*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Changement d'année

2013-2014_2 *Vendredi 24 janvier 2014*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Correction de coquilles

2013-2014_3 *Mercredi 29 janvier 2014*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Clarification des fonctions de chargement et de déchargement
— Suppression de la fonction `fraction_set`.
— Ajout du mot-clé `extern`.

2014-2015_1 *Dimanche 11 janvier 2015*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Changement d'année

2014-2015_2 *Mardi 20 janvier 2015*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Nommage des exercices

2015-2016_1 *Lundi 14 décembre 2015*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Changement d'année
— Passage au format `pandoc`
— Ajout de `cmake`

2015-2016_2 *Mardi 5 janvier 2016*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout du fichier pour le calcul du *pgcd* de 32 et 56
— Utilisation de `<stdbool.h>` pour les fonctions de chargement et de déchargement

— L'ancien exercice de création de bibliothèque n'est plus utile
— Retrait des fonctions *mutators*

2015-2016_3 *Mercredi 6 janvier 2016*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Correction de coquilles dans des fichiers `CMakeLists.txt`

2015-2016_4 *Lundi 18 janvier 2016*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Précision dans les fonctions `fraction_init_full`, `fraction_create_full` et `fraction_create_default` : la fraction devra être retournée réduite.

2015-2016_5 *Jeudi 21 janvier 2016*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Déplacement de l'utilisation de *CodeBlocks* du cahier 2 vers le cahier 1;
— Ajout du numéro de version dans la date;
— Changemement des numéros de version;
— Nommage du fichier avec le numéro de version.

2015-2016_6 *Mardi 26 janvier 2016*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Changement de nom de fonction *pgcd* vers *gcd* pour une anglicisation (*Greatest Common Divisor* en lieu et place de *Plus Grand Commun Diviseur*);
— Changement de nom de la librairie *arith* vers *arithmetic*.

2015-2016_7 *Dimanche 28 février 2016*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Réinsertion du *copyright* qui avait disparu lors du passage à la nouvelle version de *pandoc*.

2015-2016_8 *Dimanche 24 avril 2016*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Affichage du premier fichier qui était manquant ;
— Changement du langage d'un des fichiers inclus.

2015-2016_9 *Lundi 6 juin 2016*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout d'icônes.

2016-2017_1 *Mercredi 4 janvier 2017*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Changement d'année.

2016-2017_2 *Mercredi 11 janvier 2017*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Précision sur le premier `CMakeLists.txt` ;
— Précision sur la commande `make install` à lancer.

2016-2017_3 *Mercredi 18 janvier 2017*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Certaines lignes n'apparaissaient plus dans les fichiers inclus.

2016-2017_4 *Lundi 23 janvier 2017*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout d'un argument `void` à toutes les fonctions `main` sans arguments.

2016-2017_5 *Mardi 31 janvier 2017*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Explications complémentaires sur l'exercice de chargement et de déchargement ;

— Simplification des champs protégés.

2016-2017_6 *Samedi 4 février 2017*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Explications complémentaires pour l'exercice d'écriture/lecture sur fichier.

2016-2017_7 *Jeudi 9 février 2017*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout d'explications et d'options pour le lien avec entre la librairie `fraction` et la librairie `arithmetic`.

2016-2017_8 *Mercredi 15 février 2017*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout d'explications pour les variables `CMAKE_<CONFIG>_POSTFIX`.

2016-2017_9 *Mercredi 15 février 2017*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Modification de l'usage des variables `CMAKE_<CONFIG>_POSTFIX`.

2016-2017_10 *Vendredi 17 février 2017*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Simplification des fichiers `CMakeLists.txt`.

2017-2018_1 *Jeudi 9 novembre 2017*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Renommage des fonction `fraction_load` et `fraction_unload` en `fraction_init` et `fraction_finish`. Les termes ne sont donc plus *chargement* et *déchargement* mais *initialisation* et *terminaison* ;

— Renommage de la fonction `fraction_init_full` en `fraction_fill_full` pour la différencier de la fonction d'initialisation ;

— Retrait des suffixes des librairies qui complexifiait inutilement les fichiers `CMakeLists.txt` ;

— Utilisation de `pandoc 2`.

2017-2018_2 *Lundi 8 janvier 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout des figures dans la version couleur ;

— Changement des adresses email.

2017-2018_3 *Mardi 16 janvier 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Précisions sur le fichier `Makefile.default` ;
— Clarification sur les lignes à ajouter pour la compilation séparée (exercice 5 (compilation séparée)).

2017-2018_4 *Mercredi 24 janvier 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Changement du nom de l'exercice 15 (remplissage) (*initialisation* à *remplissage*) ;
— Précisions apportées pour l'exercice 14 (initialisation et finalisation).

2017-2018_5 *Jeudi 25 janvier 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Utilisation du dossier `/tmp/install` comme dossier d'installation dans l'explication de l'interface graphique de `cmake` sous *unix* (la valeur `$ENV{HOME}/.local` entrée dans le champ `CMAKE_INSTALL_PREFIX` n'était pas détectée correctement par le logiciel) ;
— précision sur le dossier `.local` du compte utilisateur.

2017-2018_6 *Mardi 30 janvier 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Précision sur l'exercice 14 (initialisation et finalisation) (merci à Robin Moreau).
— Précision sur l'exercice 15 (remplissage).

2017-2018_7 *Lundi 5 février 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Utilisation du terme *finalisation* plutôt que *terminaison*.

2017-2018_8 *Mardi 13 février 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Explication sur l'utilisation de `DESTDIR`.

2018-2019_1 *Vendredi 7 septembre 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Changement d'année.

2018-2019_2 *Jeudi 13 septembre 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Correction de coquilles dans la liste des commandes gdb.

2018-2019_3 *Mardi 18 septembre 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Précision sur l'ajout des lignes dans le fichier `CMakeLists.txt` ;
— Ajout de l'argument obligatoire (le dossier) de la commade `cmake`.

2018-2019_4 *Mardi 9 octobre 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Utilisation du style **Google** https://google.github.io/styleguide/cppguide.html pour l'indentation et autres façons de coder.

2018-2019_5 *Vendredi 12 octobre 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Utilisation d'un fichier `fraction.inc` pour accueillir la structure pour les champs protégés. Voir https://google.github.io/styleguide/cppguide.html#Self_contained_Headers.

2018-2019_6 *Dimanche 14 octobre 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Précisions sur les variables `CMAKE_INSTALL_PREFIX`, `CMAKE_PREFIX_PATH`, `CMAKE_INCLUDE_PATH` et `CMAKE_LIBRARY_PATH`.

2018-2019_7 *Mardi 6 novembre 2018*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Ajout du pointeur de fonction `fraction_realloc` dans l'exercice 23 (maîtrise de l'allocation et de la désallocation).

2019-2020_1 *Lundi 2 septembre 2019*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Utilisation du patron pandoc `eisvogel.tex` ;

2019-2020_2 *Vendredi 27 septembre 2019*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Corrections de coquilles ;

2020-2021_1 *Lundi 24 août 2020*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

— Changement d'année.

