INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

C language 2

Labs
S5 EII

2020-2021

Contents

INSA

LAB 1

Build with GCC and use the arguments of the main function

Objectives : (*1h30*)
- Discover an Unix-like environment
- Use by yourself the GCC tools to build a program ;
- Write a program using the *main* arguments ;
- Use the standard *getopt* function to handle options and parameters of a program

Requirement :
- read the course document on the build process ;
- read the course document on main parameters ;
- read the whole practical subject.

All practical sessions will be performed on a Unix-like operating system. Section 1.1 describes the set up. Section 1.2 gives basics instructions and commands for the terminal.

1.1 Virtual Machine

The graphical mode is the default nowadays on most desktop computers. In this training, we will use VirtualBox to emulate a Unix-like system (Ubuntu distribution) on our computers (Windows 7 Operating System, 64 bits). You have to add the virtual machine that has already been created. In *Raccourci_App*, launch *Oracle VM VirtualBox*. Choose "Lubuntu_prog". The others ones are for the OS courses.

By the way, Oracle VM VirtualBox beeing free, you can copy the folder containing the virtual machine if you want to work on your PC with the same configuration. It is located in `D:/Mes Documents/.VirtualBox`.

NB : you can suppress a machine from VirtualBox only if necessary but please, choose the "suppress from list" option, do not suppress files.

Now you can start your virtual machine and log on :

```
Login : user (the default user choice)
Password : user
```

1.2 Starting with the console and key commands

To continue, you will need to open a terminal window. Search for "LXTerminal". While browsing menus, you will notice that a lot of things can be done by entering commands via the keyboard. The terminal can be a more powerful tool than a mouse to handle all the tasks you will face when you know how to use it. This practical lesson presents some basic tools.

Open a terminal window. It should always show a command prompt. In French : "invite de commande", which means exactly what it is for : please give a command. Yours is as follows : *user@host:dir*. *user* will be your login name, *host* the name of the machine you are working on. The prompt is ended by $ for normal users (% for root users). Bash is the used command interpreter (i.e shell) of your terminal.

1.2.A Quickstart commands

Here are some essential commands used in the console :

Command	**Behavior**
ls	Displays a **LiS**t of files in the current working directory
cd directory$name	**C**hange the working **D**irectory
passwd	Change the **PASSW**or **D** for the current user
pwd	**P**rint **W**orking **D**irectory
man command	Display man pages (documentation) on command or function

FIGURE 1.1 – Key commands for a Linux console

You will investigate furthermore the Linux file system in OS courses.

1.2.B Some useful tips to go faster

- **History** : when you want to browse your command history (to repeat it or to edit it for example), you can use ArrowUp and ArrowDown.
- **Find a previously used command** : hit ctrl+R to search in previous commands. You can hit it until you find the one you are looking for.
- **Completion** : when typing a command, you can use the Tab key to complete a file or directory name. When multiple choices are available, the system will either signal with an audio or visual bell, or, if too many choices are available, ask you if you want to see them all.
- **Copy**/**paste** text in the terminal :
  — for a word : double-click it to select and copy it. Paste it with a middle click.
  — for a line : triple-click it to select and copy it. Paste it with a middle click.
  — for some text : select it then paste it with a middle-click.

1.2.C cd, mkdir and rm commands

Nevertheless, you have to know that each user has his own folder in the */home* directory. The content of this folder can also be viewed/modified as on a Windows OS using a file explorer ("File manager", first item on the left toolbar) of the user interface.

Try and understand the following commands. *man*, *pwd* and *ls* can help.

```
$ ls
$ pwd
$ cd /home
$ ls -la
$ cd
$ ls -la /home
$ cd ..
$ cd /
$ ls
$ cd home/user
$ mkdir tmp
$ mkdir /tmp/tete
$ cd tmp/tete
$ mkdir tmp/tete
$ cd tmp/tete
$ cd ..
$ ls
$ rm -r tete
$ ls
$ mkdir ../titi
$ rm -r ../titi
```

1.2.D Text editor

No IDE (Integrated Development Environement) as Code::Blocks is used in this practical work. In order to write your code/file, you can use *gedit* :

```
$ gedit monfichier.c &
```

If the file does not exist, it will be created. The character & enables you to use the terminal even if *gedit* is still in use (otherwise, you have to close *gedit* to use the terminal).

1.3 Build procedure using gcc

The GNU Compiler Collection provides different tools to build projects in C, C++,... For the C language, it provides the *gcc* command that compiles code and builds the program. It is widely used on Unix-like systems.

Create a *TP1* directory. Open a terminal window and create a folder named *myhello-dir* in your *TP1* directory. Go to this latter directory and create a file named *myhello.c* which displays "Hello World !". Then execute to compile the code :

```
$ gcc -c myhello.c
```

Using the *ls* command, look at the generated file(s). To create the executable file (the linking step), type :

```
$ gcc myhello.o -o myhello
```

Run the *myhello* executable file :

```
$ ./myhello
```

It is possible to combine both steps (compilation + linking) in a unique command :

```
$ gcc myhello.c -o myhello
```

1.4 The *main* arguments

Open a terminal window and create a folder named *getopt-dir* in your *TP1* directory. Go to this latter directory.

The prototype of the *main* function can be of various forms, for example :

```
void main();
int main();
int main(int argc, char *argv[]);
int main(int argc, char *argv[], char **arge);
```

with :

- *argc* : number of arguments of the program
- *argv* : array of the arguments
- *arge* : array of the inherited environment variables. A NULL pointer ends the list.

argv[0] is always the name of the program, so argc >=1.
An argument is a string delimited by a space character.

The arguments of a program are very useful for the operating system (OS) to know the run-time context and the options that will impact the program behaviour.

1.4.A *echo* command

Type in the terminal :

```
$ echo Salut toi
```

Here is a program *myecho.c* that does the same thing, ie displays the arguments of the program, but in blue [1] :

```
#include <stdio.h>
int main (int argc, char *argv[])
{
  int i;
  for (i=1 ; i<argc ; ++i)
    printf("\033[34m%s\033[0m%c",argv[i], ( i< argc-1 ) ? ' ' : '\n');
  return 0 ;
}
```

Type the following command to build the executable file *myecho* and test it [2] :

```
$ gcc myecho.c -o myecho
$./myecho Salut toi
```

1.4.B Prints the environment variables

Modify your program to also display the environment variables of your program.

1. for the colour printing, you will study this further in OS course, but here is a simple explanation. 033[means the next characters are not characters to be printed, the following 34 (resp. 0) is the blue colour code (resp. default colour) and the character m means it is the end of the non-printable characters
2. You have to say that your program in the current directory, so prefix it with "./". You will see why in OS course.

1.4.C Program option

Some options can be defined to set the behaviour of a program. They usually use the '-' character. For example, the *ls* command provides a more complete description with the "-l" option. It can be usually located anywhere in the argument list :

```
$ ls -l
$ ls Desktop -l
$ ls -l Desktop
```

To detect your program option, you can do it yourself by scanning all your argument list and compare each argument to a possible option. It can become very painful if you have many options, all the more that you will have to handle options for every program. As it is a very common task, a function is provided by GCC : *getopt*[3].

Here is its prototype :

```
int getopt (int argc, char **argv, char *options)
```

where :

- *argc* is the *argc* argument of the main function, ie its number of arguments.
- *argv* is the *argv* argument of the main function, ie its list of arguments.
- *options* are the options whose presence is to be checked. If the option is supposed to be followed by a value, its character name is followed by the character ' :'. For example, if the three options "-a -b -c value_for_c_option" are possible for a program, *options* will be set to "abc:"

Normally, *getopt* is called in a loop as shown in the example on the next page. When *getopt* returns -1, indicating no more options are present, the loop terminates.
A switch statement is used to dispatch on the return value from *getopt*. In typical use, each case just sets a variable that is used later in the program.
A second loop is used to process the remaining non-option arguments. Indeed, *getopt* changes the position of arguments in *argv* and put all the arguments associated to the options at the beginning of *argv* and the other ones after.

This process is based on some variables associated to the use of *getopt* :

- char * *optarg* : set by *getopt* to point at the value the option value, for those options that accept values.
- int *optind* : once *getopt* has found all of the option arguments, you can use this variable to determine where the remaining non-option arguments begin.
- int *opterr* : If the value of this variable is nonzero, then *getopt* prints an error message to the standard error stream if it encounters an unknown option character or an option with a missing required argument. This is the default behavior. If you set this variable to zero, *getop*t does not print any messages, but it still returns the character ? to indicate an error.
- int *optopt* : when *getopt* encounters an unknown option character or an option with a missing required argument, it stores that option character in this variable. You can use this for providing your own diagnostic messages.

3. http://www.gnu.org/software/libc/manual/html_node/Getopt.html

Here is the example of use :

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

int main (int argc, char **argv)
{
  int aflag = 0;
  int bflag = 0;
  char *cvalue = NULL;
  int index;
  int c;

  opterr = 0;
  while ((c = getopt (argc, argv, "abc:")) != -1)
    switch (c)
      {
      case 'a':
              aflag = 1;
              break;
      case 'b':
              bflag = 1;
              break;
      case 'c':
              cvalue = optarg;
              break;
      case '?':
              if (optopt == 'c')
                fprintf (stderr, "%s : option -%c requires an argument.\n",
                    argv[0], optopt);
              else if (isprint (optopt))
                fprintf (stderr, "%s : unknown option `-%c'.\n", argv[0],
                    optopt);
              else
                fprintf (stderr,"%s : unknown option character `\\x%x'.\n",
                    argv[0], optopt);
              return 1;
      default:
          abort ();
      }

  printf ("aflag = %d, bflag = %d, cvalue = %s\n", aflag, bflag, cvalue);

  for (index = optind; index < argc; index++)
    printf ("Non-option argument %s\n", argv[index]);
  return 0;
}
```

This code is provided in a file *test-getopt.c*, test it :

```
$ gcc test-getopt.c -o test-getopt
$./test-getopt Salut toi
$./test-getopt Salut toi -a
$./test-getopt -a Salut  -c Bonjour toi -b
```

Modify this file using your code in *myecho.c* to display the argument in blue if option -b is used, in default colour otherwise. Test it. Add then an option -c such as :

— if *-c lower* is passed in the arguments of the program, all the letters printed by the program are in lower case

— if *-c upper* is passed in the arguments of the program, all the letters printed by the program are in upper case

INSA

LAB 2

Makefile and Stack structure

Objectives : (*1h30*)
- Handle the build steps with GCC in a Makefile ;
- Study a stack of int ;
- Code a function with an ellipsis ;
- Study and practice basic unit testing.

Requirement :
- Read the course document on Makefile ;
- Read the course document on ellipsis ;
- Read the whole practical subject.

Be careful, the stack implementation be be done in this lab is required for the next lab session.

Create a *TP2* directory.

2.1 Makefile

Open a terminal window and create a folder named *makefile-dir* in your *TP2* directory. Go to this latter directory and copy in it the provided *myhello.c* file.

2.1.A Presentation

Make is a program that manages the build process of targets (as object files, executable files, libraries,...) according to their dependencies (as header files, code files,...). It is widely used to build and install complex projects. The idea is to exploit the similarities of the building process of each file. The basic rule is always the same and can be used without any configuration file for very simple program.

For example, remove your *myhello* and *myhello.o* binary files and run the *make* command :

```
$ rm myhello myhello.o
$ make myhello
```

What build rule is used ?

For most complex projects (i.e. all realistic projects), the building rules have to be described in a configuration file named Makefile by default (be careful, it is case-sensitive). For each target to be made (for example the binary *myhello*), you will find a rule in the Makefile. We will explain what is precisely a rule using the following example.

2.1.B Basic Makefile

Download ans uncompress the provided textitmakefile_basic zip archive. It contains a file named *mytools.c* in which is implemented the function *void myHello()*. The file *myhello.c* calls this function. The header file *mytools.h* is included in both files.

In a Makefile, a **rule** starts with the name of the **target** (for example, *myhello*) which is followed by its **dependencies**. For example *myhello* depends on *myhello.o* and *mytools.o*. It means that the binary *myhello* is out-of-date if one of its dependencies is newer than it : it needs to be rebuilt. It also means that it should be rebuilt ONLY if these two files changed. The lines following the target dependencies description contain the command(s) to be run if at least one of the dependencies is newer than the latest version of the target. These lines have to **begin with tabulation**.

For this project, the Makefile is then :

```
myhello: myhello.o mytools.o
    gcc myhello.o mytools.o -o myhello

myhello.o: myhello.c mytools.h
    gcc -Wall -c  myhello.c

mytools.o: mytools.c mytools.h
    gcc -Wall -c  mytools.c
```

Create the Makefile with this text.

Generate the *myhello.o* file by typing in the terminal window :

```
$ rm myhello.o myhello
$ make myhello.o
```

Generate the executable file by typing :

```
$ make myhello
```

Let's note that if you type only *make* without any argument, the default target is the first one. This is why the rule to build the binaries is the first one usually. As the executable is the first one in our case, just typing *make* is enough to build the program.

The common targets in a Makefile also include a clean process (be careful, do not add a space after the * character : you will suppress every file of your directory).

```
clean:
    rm -f *.o *~
    rm -f myhello
```

Describe in detail what does this additional rule mean. Test it :

```
    $ make clean
```

2.1.C Usual Makefile

As every Makefile looks like another and rules are similar, some variables have been added to make writing Makefile easier :

$@	The target name
$<	The first dependency name
$^	The dependencies list
$?	The list of dependencies that have been more recently modified than the target
$*	The target name without its suffix

FIGURE 2.1 – Main make variables

In order to maintain easily the Makefile, you also declare variables at the beginning of the Makefile to configure the building process. The very basic ones are :

```
CC=gcc              #Program for compiling C programs
CFLAGS=-Wall        #Extra flags to give to the C compiler
LDFLAGS=            #Extra flags to give to the linker
```

The *myhello.o* rule becomes with such a change :

```
myhello.o: myhello.c mytools.h
    $(CC) $(CFLAGS) -c $< -o $@
```

Using the variables is very useful if you want to build using different configuration : you only have to change your variable and not every rule. For example, when debugging, you change CFLAGS to CFLAGS =-Wall -d and this will apply to each target.

You can also note that the rules to build *myhello.o* and *mytools.o* are very similar and written the same way. You can use patterns [1]. It is very useful since you do not have to write a rule for each .o added in your project. For example, replace the rules *myhello.o* and *mytools.o* by :

```
myhello.o: mytools.h
mytools.o: mytools.h

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

Modify every rule with these changes and test it.

1. http://www.gnu.org/software/make/manual/html_node/Pattern-Rules.html

2.2 A stack

A stack is a container where the last data inserted is the first to be removed. The last inserted value is defined as the top of the stack. It can be implemented using an array. The top of the stack is represented by the index of the last inserted value. This index increases when a new value is added (push operation) and decreases when a value is removed (pop operation).

A stack is useful in many problems where data is managed with a LIFO strategy (Last In, First Out). One of them will be the evaluation of postfix expressions. Postfix expressions will be introduced latter. The focus is first on the stack implementation.

As regards the development environment in this lab session, no IDE is permitted. Edit your code in a text editor and build the program file using *gcc* and *make* commands as required.

2.3 Implementation of a stack of int

An implementation of a stack of int is provided ("stack.c/.h") with basic unit tests on the provided functionalities in the function *test_stack*.

Build the program. It depends on three objects files ("main.o", "stack.o" and "test.o") :

```
$ gcc -c test.c -o test.o                    #compiling test.c
$ gcc -c stack.c -o stack.o                  #compiling stack.c
$ gcc -c main.c -o main.o                    #compiling main.c
$ gcc  main.o stack.o test.o -o main         #linking
$ ./main                                     #running
```

As you can see, it is easily to see what is tested and if it succeeded without having to read the source file of the test function, in the spirit of what has been introduced in the course. For this, we used the test function provided in the module "test". You can also find the code the Appendix B.

In order to compile only the binary files associated to the modified source files and not to write each time all these commands to build the program, we will use a Makefile. Write the Makefile to test the module. Test your Makefile :

```
$ rm *.o main
$ make                                       #build (compilation + linking)
$ ./main                                     #running
```

It is time to start coding. Write the following function to add several values to the stack in one go :

```
int fill_stack(Stack *s, int n_element, ...);
```

It should return 1 if all values were correctly added, 0 otherwise.

Write also the test code for *fill_stack* the *test_stack* function the same way the other tests were processed. As it is important the test the generic behavior but also specific cases, test it with 4 values to be added (*n_element* = 4) and with no value added (*n_element* = 0).

Then build using the provided Makefile your program calling this function :

```
$ make                                       #build (compilation + linking)
```

Study the build process to see what rule is called and in what order.
Then test your program :

```
$ ./main                                    #running
```

ANNEXE A

Doxygen

A.1 Présentation

C'est un outil d'auto-documentation pour le C, C++, Java, Objective-C, Python, IDL, Fortran, VHDL, PHP, C#. Au moyen de balises placées autour du code et permettant de décrire fonctions/classes/structures/..., il est possible de générer une documentation technique sous plusieurs formats (html,xml,..) [1].

A la base, **Doxygen** est un utilitaire qui parcourt des fichiers sources et génére des pages de documentation en fonction de la configuration voulue par l'utilisateur. Cette configuration est stockée dans un fichier **Doxyfile** qui contient beaucoup de variables, pas forcément lisible pour un non-initié. C'est pourquoi nous allons utiliser **Doxywizard**, un GUI (Graphical User Interface) qui permet de gérer plus facilement la configuration et le lancement de Doxygen.

Un manuel, aide et exemples sont disponibles à partir de :

`http://www.stack.nl/~dimitri/doxygen`

A.2 Installation

A.2.A Doxywizard

Télécharger l'outil nécessaire à l'adresse suivante et suivre les instructions :

`http://www.stack.nl/~dimitri/doxygen/download.html`

A.2.B graphviz

Éventuellement, on peut aussi installer **graphviz** si on veut faire apparaître des graphes d'appels ou de classes dans la documentation (surtout utile pour la POO).

`http://www.graphviz.org/Download..php`

1. Le principe est assez similaire à Javadoc, outil d'auto-documentation pour Java

A.3 Commenter le code

Le principe est de mettre des commentaires respectant une certaine syntaxe pour pouvoir être analysés.

Les blocs commentaires qui seront interprétés par Doxygen peuvent commencer et finir de plusieurs façons :

```
/**
* Commentaires Doxygen
*/

/*!
* Commentaires Doxygen
*/

///
/// Commentaires Doxygen
///

//!
//! Commentaires Doxygen
//!
```

La syntaxe des blocs à base de balises permet à Doxygen de connaître la nature du code commenté et les différents paramètres qui vont avec. Attention, si la syntaxe est incorrecte ou incomplète, Doxygen n'inclura pas la documentation associée dans le fichier final (penser à regarder dans le fichier log pour voir les messages issus de la génération de la documentation).

Voici les blocs les plus utiles, pour les autres, voir la documentation complète sur le site web. Un exemple est fourni par la suite, ainsi que la trame d'un fichier de base.

Nature du bloc	Syntaxe Doxygen

Fichier	`/**`
	` * \file nom du fichier`
	` * \brief brve description de son contenu`
	` * \author nom prnom`
	` * \version X.Y`
	` * \date date de cration`
	` *`
	` * Description plus dtaille du contenu du fichier`
	` *`
	` */`

Fonction	`/**`
	`* \fn prototype de la fonction`
	`* \brief description brve de la fonction`
	`*`
	`* Un peu plus de dtails sur la fonction`
	`*`
	`* \param[in] nom description du paramtre lu mais pas modifi.`
	`* \param[out] nom description du paramtre modifi mais pas lu.`
	`* \param[in,out] nom description du paramtre lu et modifi.`
	`*`
	`* \return le type de retour et sa description`
	`*/`

Macro	`/**`
	`* \def nom de la macro`
	`* Description de la macro`
	`*/`

Structure / Classe / Énumération /...	`/**`
	`* \struct nom de la structure`
	`*  (\class pour une classe,`
	`*  \enum pour un type numr, ...)`
	`* \brief brive description de l'objet`
	`*`
	`* Description dtaille de l'objet`
	`*/`

Champ / membre d'une structure / classe / énumération /...	`/*!< Commentaire sur le champ/membre de la structure/classe */`

A.4 La génération de documentation

Sous Doxywizard, il y a deux modes de configuration de la génération de la documentation, Wizard et Expert. Les deux mettent à jour le Doxyfile mais le premier est plus simple, pour les débutants. Le deuxième permet de personnaliser la documentation et nous verrons quelques variables intéressantes pour le C.

A.4.A Configuration Wizard

Il y a 4 parties à renseigner :

- Project

  Les informations permettent d'identifier le projet, où sont les sources, où mettre la documentation résultante.

- Mode

  En ce qui concerne le mode d'extraction, "Documented entities only" et "All Entities" permettent de définir quels éléments du code doivent être commentés et référencés dans la documentation finale. Pour un réglage plus fin, il faut mettre les mains dans le cambouis en Expert.
  La case "Include cross-referenced source code in the output" est bien pratique, elle permet de faire des aller-retour entre la documentation et le code facilement.
  En ce qui concerne le type de programme analysé, choisir le langage utilisé dans les fichiers sources.

- Output

  Choisir le format de sortie (HTML en général, il supporte plus d'options que les autres modes).

- Diagrams

  Indiquer si l'on veut des diagrammes ou pas. Pour la 3ème option, il faut avoir installer graphviz. Elle peut ne pas marcher(le code des graphes s'affiche en dur alors qu'on voudrait un joli dessin), dans ce cas, se reporter au paragraphe sur le sujet "Dot" dans la configuration Expert.
  Dans les graphes générés, on peut voir l'interaction entre classes/structures, les dépendances de fichiers,etc. Les call graphs ou called by graphs permettent de voir qui appelle quoi, ce qui peut être très lourd suivant le code.

A.4.B Configuration Expert

Toutes les variables du Doxyfile peuvent être modifiées ici. Il suffit de passer la souris sur le nom de l'une d'entre elles pour voir la description. Nous allons en citer certaines très utiles au fil de la description des différentes catégories :

- Project

  Toutes les variables permettant de décrire l'environnement du projet.
  Si vous avez sélectionné dans la configuration Wizard une optimisation pour le langage C, la variable OPTIMIZE_OUTPUT_C devrait être activée.

  Autres variables intéressantes :
  — BRIEF_MEMBER_DESC et REPEAT_BRIEF qui permettent de configurer l'emplacement des descriptions brèves dans la documentation
  — OUTPUT_LANGAGE : on peut choisir "French" pour avoir des onglets et des liens en français.

- Build

  Les variables configurent ici l'extraction de la documentation : quels éléments doivent être documentés et référencés, le tri des informations, etc.

- Messages

  Il s'agit de la configuration de la génération du fichier de log, pour avoir un Doxygen plus ou moins bavard.

- Input

  Il est possible d'affiner ici la spécification des fichiers scannés par Doxygen. Typiquement, s'il y a autre chose que des fichiers sources dans l'arborescence, autant le signaler.
  INPUT_ENCODING est une variable utile pour un bon affichage. En effet, si on ne met pas le bon encodage des fichiers sources et qu'il y a des caractères un peu particuliers (accents par exemple) dans la documentation, on se retrouvera avec une documentation pas très agréable à regarder. Par défaut c'est UTF-8 mais essayer ISO-8859-1 éventuellement si vous avez des soucis d'affichage d'accents.

- Source Browser

  Ces variables permettent de configurer les fichiers sources générés en HTML et les références correspondantes dans/vers la documentation.

- Index

  Activer ALPHABETICAL_INDEX permet d'avoir un index des classes/structures par ordre alphabétique.

- HTML, LaTeX, RTF, Man, XML

  On a la possibilité de personnaliser le style des documents en sortie (cadres HTML par exemple).

- Dot

  La génération des graphes se paramètre ici. On retrouve les différents graphes pouvant être générés.
  Si on veut utiliser graphviz, s'assurer que dans DOT_PATH se trouve le bon chemin du genre ".../graphviz/bin".

A.4.C Run

Une fois que vous avez tout renseigné suivant l'une des deux configurations, vous pouvez sauvegarder vos choix. Cela crée un Doxyfile et cela permettra de le recharger plus tard.

Si vous avez effectué un changement de configuration, il est des fois nécessaire d'effacer les fichiers générés précédemment pour que la nouvelle configuration fonctionne comme elle le devrait.

Ensuite, lancer la génération avec le bouton "Run Doxygen". Le fichier de log s'affiche, vérifier qu'il n'y a pas d'erreurs et vous pouvez ensuite visualiser la sortie en HTML directement.

A.5 Exemple

Voici un fichier source et la documentation correspondante générée par Doxygen.

A.5.A Fichier source

```
/**
* \file example.c
* \brief Programme de tests.
* \version 1.0
* \author Muriel Pressigout
* \date 23 mars 2009
*
* Programme d'exemple pour doxygen : manipulation d'une structure
*
*/

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


/**
* \def TAILLEMAX
* Dfinit la taille maximum d'une chane de caractres
*/
#define TAILLEMAX 50


/**
* \struct personne
* \brief Structure reprsentant les donnes d'une personne
*
* Une personne sera dsigne par son prnom, ge, un conjoint et ses enfants.
*/
typedef struct
{    char prenom[TAILLEMAX] ;        /*!< Chane de caractres pour le prnom.  */
     int age ;                       /*!< Age de la personne. */
     struct personne * conjoint ;    /*!< Pointeur vers le conjoint
                                     (NULL si clibataire).*/
     struct personne ** enfants ;    /*!< Tableau de pointeurs vers les enfants
                                     (NULL si sans enfants).*/
}
personne;


/**
* \enum err_personne
* \brief Constantes d'erreurs.
*
* err_personne est une srie de constantes prdfinie pour
* la manipulation de la structure personne.
*/
typedef enum
{    PERSONNE_NO_ERR,            /*!< Pas d'erreur. */
     PERSONNE_MEMORY_ERR,        /*!< Problme d'allocation mmoire. */
     PERSONNE_INIT_ERR,          /*!< Objet mal initialis. */
     PERSONNE_SIZE_ARRAY_ERR   /*!< Problme sur le dimensionnement
                                 des chanes de caractres. */
}
err_personne;
```

```
/**
* \fn err_personne personne_new(char * nom, char * prenom, int age, personne *
    p)
* \brief Fonction de cration d'une nouvelle instance d'une structure personne.
*
* Aprs l'appel de cette fonction, si tout s'est bien pass,
* les champs nom, prenom, age ont la valeur passe en paramtre.
* Les champs conjoint et enfants sont initialiss  NULL.
*
* \param[in] nom Chane contenant le nom de la personne.
* \param[in] prenom Chane contenant le prenom de la personne.
* \param[in] age Entier reprsentant l'ge de la personne.
* \param[out] p Pointeur vers cette nouvelle instance.
* \return Un code d'erreur de type err_personne.
*/
err_personne personne_new(char * nom, char * prenom, int age, personne * p)
{
    int taille_nom, taille_prenom;

    //initialisation du champ prenom
    if (prenom == NULL)
        return PERSONNE_INIT_ERR;

    taille_prenom = strlen (prenom);

    if (taille_prenom >= TAILLEMAX)
        return PERSONNE_SIZE_ARRAY_ERR;

    strcpy(p->prenom, prenom);

    //initialisation du champ age
    if (age < 0 )
        return PERSONNE_INIT_ERR;

    p->age = age;

    //initialisation du champ conjoint
    p->conjoint = NULL;

    //initialisation du champ enfants
    p->enfants = NULL;

    return PERSONNE_NO_ERR;
}


/**
* \fn err_personne personne_delete(personne * p)
* \brief Fonction de destruction d'une instance d'une structure personne.
*
* Cette fonction dsalloue la mmoire alloue au tableau des pointeurs vers les
    enfants,
* ne supprime pas les donnes des enfants ni du conjoint.
* Les nom et prnom sont vides et l'ge gal  -1.
*
* \param[in,out] p Pointeur vers cette instance.
* \return Un code d'erreur de type err_personne.
*/
err_personne personne_delete(personne * p)
{
    strcpy(p->prenom, "");
```

```
    p->age=-1;

    p->conjoint= NULL;

    if(p->enfants != NULL)
    {
        free(p->enfants);
        p->enfants=NULL;
    }

    return PERSONNE_NO_ERR;
}


/**
* \fn int main (void)
* \brief Entre du programme.
*
* \return EXIT_SUCCESS - Arrt normal du programme.
*/
int main (void)
{
    personne p1;
    err_personne code_retour;

    code_retour = personne_new("Dupont", "Dupond", 50, &p1);

    if(code_retour!=PERSONNE_NO_ERR)
    {
        fprintf(stderr,"Problme lors de personne_new(...) : retour %d\n");
        fprintf(stderr," ****FIN DU PROGRAMME\n",code_retour);
        personne_delete(&p1);
        return -1;
    }
    else
        fprintf(stdout,"Fin de la cration de l'instance personne\n");


    //pleiiiiiin de code

    code_retour = personne_delete(&p1);

    if(code_retour!=PERSONNE_NO_ERR)
    {
        fprintf(stderr,"Problme lors de personne_delete(...) : retour %d\n");
        fprintf(stderr," ****FIN DU PROGRAMME\n",code_retour);
        return -1;
    }
    else
        fprintf(stdout,"Fin de la suppression de l'instance personne\n");

   return EXIT_SUCCESS;
}
```

A.5.B Documentation générée en HTML

Sur les Figures A.1, A.2 et A.3 sont présentées trois vues de ce que donne Doxygen en HTML sur l'exemple fourni.

La navigation entre les différents éléments documentés est facile, il y a des liens pour

passer d'une documentation à l'autre, ou encore au code correspondant, ou inversement, en cliquant sur une ligne du code, à la documentation associée à cette ligne.

Le code présenté est nettoyé des balises Doxygen, ce qui clarifie la lecture du fichier source.

A.6 Trame de base

Voici une trame pour un fichier contenant le programme principal.

```
/*!
 * \file  ...
 * \brief ...
 * \author ...
*/


#include <stdlib.h>           /* EXIT_SUCCESS */


/*!
 * \fn int main(void)
 *   ...
 * \return EXIT_SUCCESS si tout s'est bien pass
 */
int main(void)
 {

  return EXIT_SUCCESS;
} /* fin main */
```

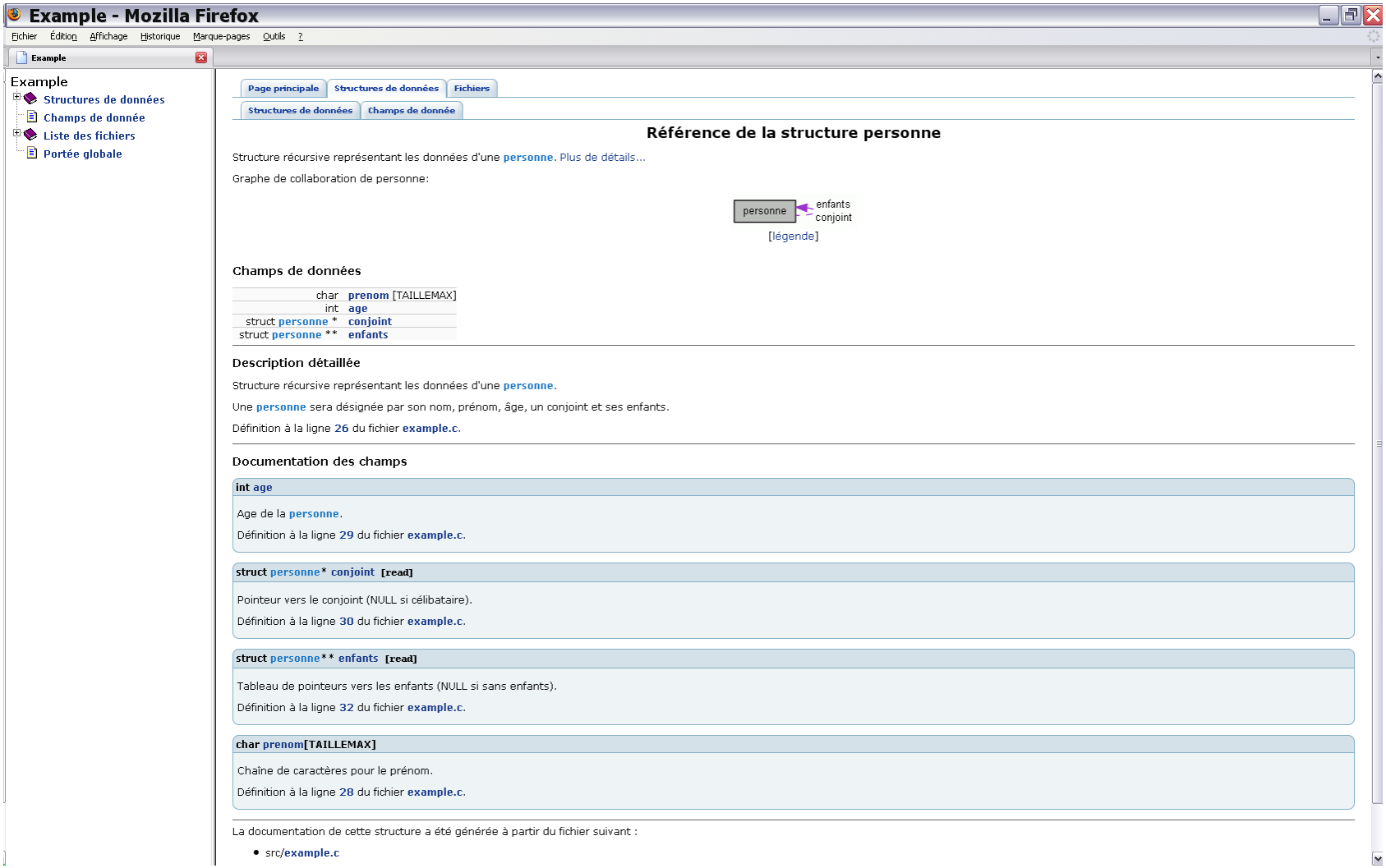

FIGURE A.1 – Documentation d'une structure en HTML généré avec Doxygen

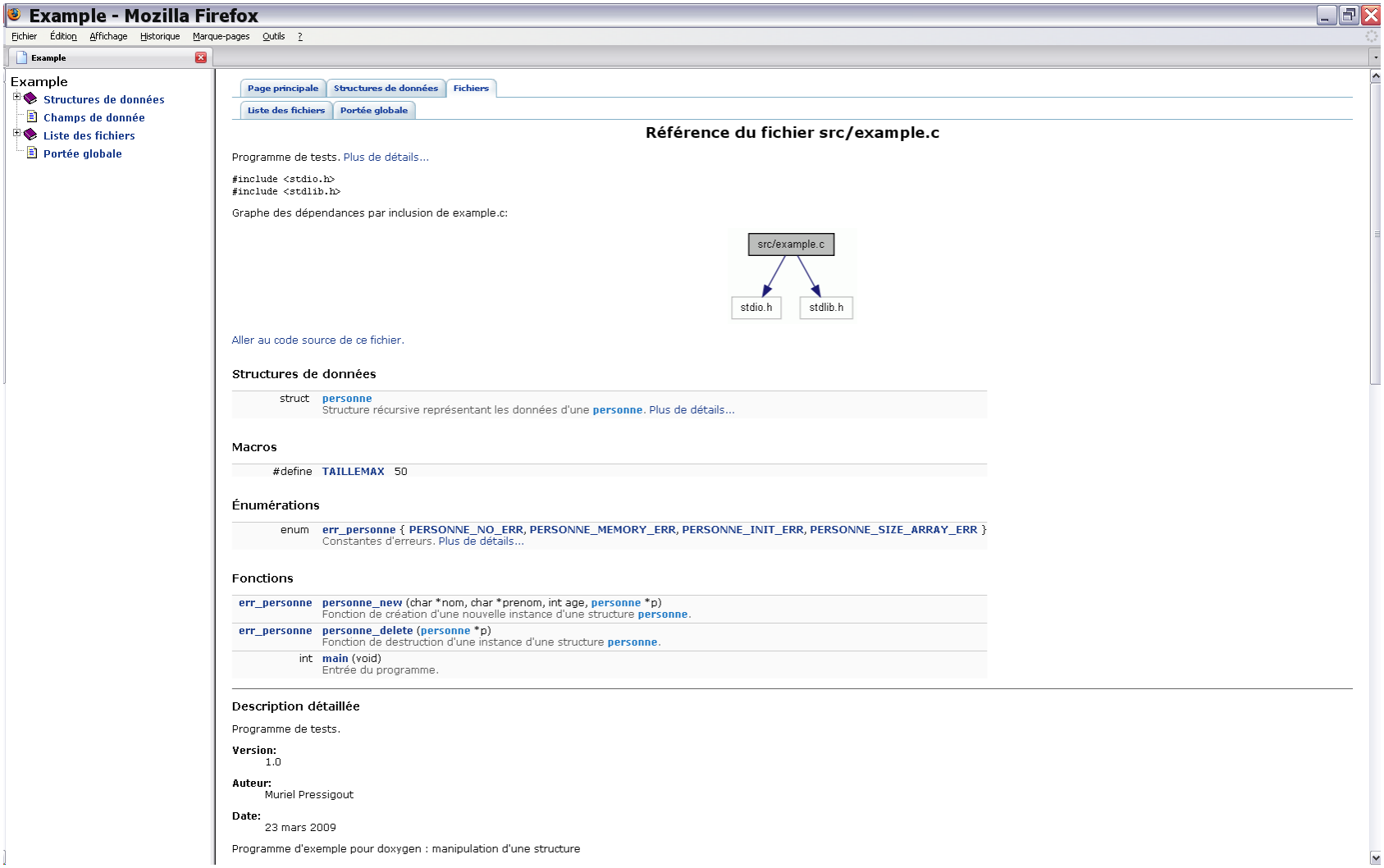

FIGURE A.2 – Documentation d'une structure en HTML généré avec Doxygen

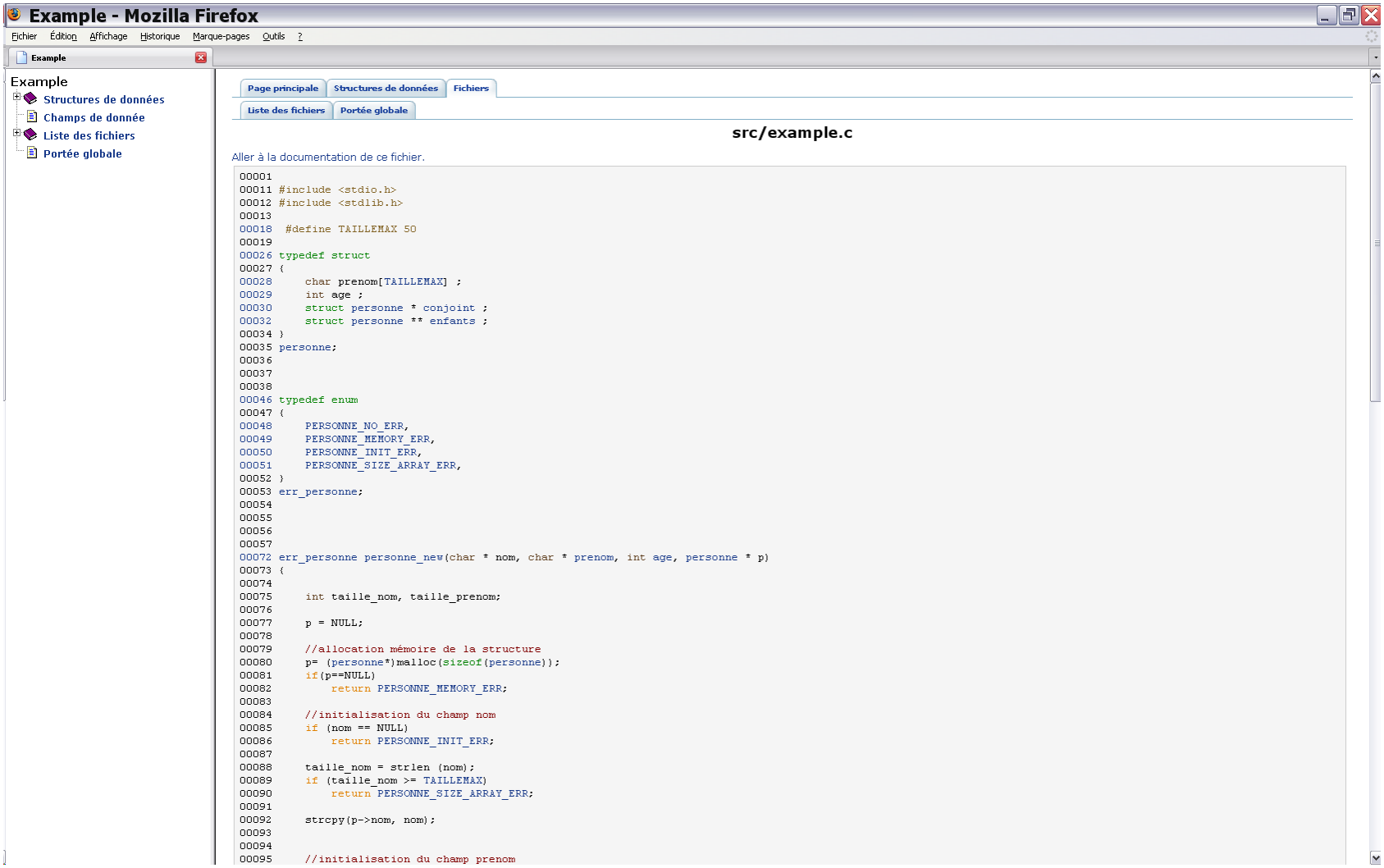

FIGURE A.3 – Contenu d'un fichier source en HTML généré avec Doxygen

ANNEXE B

Unit test functions

When performing unit tests, it is required to display what functionnality is tested, in what case and if it has passed or not. We provide the following code to do it for simple cases, without using a specific tool. You can extend it for other types (double, float, char, structures, enum types,...).

```
/*!\file test.c
* Basic unit testing functions
*/

#include <stdio.h>
#include <string.h>

/*!
* \brief Displays a test whose result has to be checked by the user.
* The test message is displayed in blue.
* \param[in] message the message indicating what it is tested
*            and what value should be expected
*/
void display_test_check_by_user(char * message)
{
  printf("\033[36m%s\033[0m :\n",message);
}

/*!
* \brief Displays the result of a test returning an int
* The test message is displayed in green is the test has passed, in red
    otherwise.
* \param[in] message the message indicating what it is tested
* \param[in] expected_value the expected value
* \param[in] expected_value the tested value
*/
void display_test_int(char * message, int expected_value, int obtained_value)
{
  (expected_value == obtained_value)?
    fprintf(stdout,"\033[32m%s : %s\033[0m\n",message,"PASSED"):
    fprintf(stdout,"\033[31m%s : %s\033[0m\n",message,"FAILED");
}

/*!
* \brief Displays the result of a test returning a string
* The test message is displayed in green is the test has passed, in red
    otherwise.
* \param[in] message the message indicating what it is tested
* \param[in] expected_value the expected value
* \param[in] expected_value the tested value
*/
void display_test_string(char * message, char * expected_value, char *
    obtained_value)
{
  (strcmp(expected_value,obtained_value) == 0 )?
```

```
        fprintf(stdout,"\033[32m%s : %s\033[0m\n",message,"PASSED");
        fprintf(stdout,"\033[31m%s : %s\033[0m\n",message,"FAILED");
    }
```

INSA Rennes
20 Avenue des Buttes de Coësmes
CS 70839
35708 Rennes Cedex 7

Tél. +33 (0) 2 23 23 82 00
Fax +33 (0) 2 23 23 83 96

www.insa-rennes.fr

INSA

UNIVERSITÉ EUROPÉENNE DE BRETAGNE

Commission des Titres d'Ingénieur

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE