LABS 2

Unit tests and code coverage

Objectives : (*1h30*)
- Write unit tests for C code
- Manipulate CMocka framework
- Measure the code coverage of the unit tests with *gcov*
- Analyse the code using valgrind

This lab is designed for a Unix-like OS. Please log on the 3EII virtual box associated to your group. The root password is *user*.
As you will have to install tools, first update the package versions :

```
$ sudo apt-get update
```

and the basic build tools if needed:

```
$ sudo apt-get install build-essential
```

2.1 Unit testing with CMocka

2.1.A Overview of this section

You will :

- download and build the counter module 2.1.B
- install the CMocka library as described in subsection 2.1.C
- provide the first test for this module using CMocka 2.1.D
- study the code coverage for this test using *gcov/lcov* 2.1.E
- use a Makefile to launch all the tools seen in this lab 2.1.F
- launch 2.1.G

2.1.B The counter module

First download from the Moodle site the "counter.zip" archive. Create a CodeBlocks project including those files, build it and run it.

2.1.C Install CMocka

Create a directory named "CMocka" and download in this directory the "cmocka-1.1.5.tar.xz" archive from the download sectionof the CMocka site `https://cmocka.org/`. Go the "CMocka" directory.

Decompress the file by typing :

```
$ tar xf cmocka-1.1.5.tar.xz
```

A directory named "cmocka-1.1.5" is now there. It contains the source code of the CMocka library. You need to build it to use it in your project. CMocka library use the CMake build tool to generate project configuration files (*.sln* Visual Studio solution configuration, *.cbp* codeblocks project, Makefiles, ...) to build it with respect to your environment (OS, compiler,...). By default, it generates Makefiles, which is convenient for us.

As it is preferable to set binary files apart from source code, follow the next instructions to build the CMocka library.

First check you are in the "CMocka" directory and that there is the "cmocka-1.1.5" directory inside. Then type :

```
$ mkdir cmocka-build
$ cd cmocka-build
$ cmake ../cmocka-1.1.5
$ make
$ sudo make install
```

You will find the explanation of these commands in appendix A.

Your CMocka library is now built. You can use it now.

2.1.D Unit testing of the counter module : test the counter incrementation

You have to test this counter module. You will write in the file "UTest-counter.c" the test for the counter.

To use CMocka in your project, you need to:

- include the following headers in "UTest-counter.c":

```
#include <stdarg.h>
#include <stddef.h>
#include <setjmp.h>
#include <cmocka.h>
```

- indicate to *gcc* you are using functions of the CMocka library. We only have to add the library name to the linker since it has been installed in a location where *gcc* can find it. To add the library name, we have to add to the linker option "-lcmocka". In a Makefile, it would be added in the LDFLAGS variable.

The first part is to test the counter incrementation. As we want to be sure that an occurring error comes only from the incrementation code, the counter needs to be set up **before** the test function. To test this function you then need to code :

- a setup function *setup_zero* that creates a counter with a value equals to 0. This counter will be transmitted by the parameter *state* as seen during the lesson. This function prototype follows the rule for the CMocka setup functions :

```
static int setup_zero(void **state);
```

Remember that a CMocka setup function should return 0 if everything was OK, -1 otherwise, in order to prevent the test to be run if the setup was wrong.

- the test function to test the increment of the counter. Its prototype follows the rule for the CMocka setup functions :

```
static void test_increment(void **state);
```

- as the setup function performs a dynamic allocation of a counter, you need a teardown function *teardown* to free the memory. Its prototype follows the rule for the CMocka teardown functions :

```
static int teardown(void **state);
```

Code those three functions and the main function running the test. **If you have an issue when running your program with the CMocka shared library (.so), read the appendix B**.

2.1.E Measure your test code coverage

Check using *gcov.lcov* the code coverage of your test. See the appendix C to see how to build your project so it generates data for *gcov/lcov* and how to read the files provided by *gcov/lcov*.

2.1.F Unit test with a mock function

The function *s*et_counter_rand calls the function *r*and. To be able to check its behavior, we need to know what the *r*and function will return. As it is not possible, we will use a mock function to simulate and wrap the function*r*and. To do so, you will have to :

1. code the function *__wrap_rand* which should have the same prototype than *r*and. It will return the *i*nt value returned by the macro *m*ock provided by the CMocka
2. code the test of *s*et_counter_rand using the macro *w*ill_return of CMocka to specify the behavior of the mocked object
3. use the wrap option of the *g*cc linker, *l*d, you need to add AT THE BEGINNING of the linker options :

```
-Wl,--wrap=rand
```

Remark: there is no space before or after the comma.

Run the test and check the result (console output + *g*cov/lcov).

2.1.G Full setup for tests

Test the provided Makefile. It launchs the unit tests, the tests of the tests (code coverage and valgrind). For the code coverage, html output is generated. Open the *i*ndex.html file in the *out* directory to see the results.

2.2 Code analysis with Valgrind

You will study the output of the main tool of Valgrind, Memcheck. The provided code is similar to the code you have tested with your memory leak detector last semester.

2.2.A Install Valgrind

Type :

```
$ sudo apt-get install valgrind
```

It is done !

2.2.B Use Valgrind

First go in the provided "Valgrind" directory and build your program using the provided Makefile in the debug mode :

```
$ make DEBUG=yes
```

Then run Valgrind on the program with the option to get as much as possible information about memory leaks:

```
$ valgrind --leak-check=yes ./test-valgrind
```

There are errors in the code. Analyse the output of Valgrind to check how it detects those errors. Fix them.

ANNEXE A

Details of the CMocka library installation

To install CMocka, you have typed in 2.1.C:

```
$ mkdir cmocka-build
$ cd cmocka-build
$ cmake ../cmocka-1.1.5
$ make
$ sudo make install
```

Here is an explanation of what you have done :

- Separation of library sources and library binaries
- The *cmake* command generates Makefiles associated to the CMake configuration find in "../cmocka-1.1.5".
- The *make* command build the library according to the Makefiles.
- The *make install* installs properly the library on the computer (headers in "/usr/local/include" and library file "libcmocka.so" in "/usr/local/lib"). Those locations are default locations where *gcc* will look for headers and libraries. So you do not need to add information to compiler and linker configurations to find them. *sudo* is to get the root privileges as you install something in a protected area.

ANNEXE B

Update dynamic libraries directories

If the OS does not find a shared library (*.so on Linux) when running a program, after checking it is installed, you may have to update the location where the OS find the dynamic libraries. An example will be provided for the CMocka library however the solution works with any dynamic library installed somewhere the OS does not find it[1].

B.1 Here is an explanation of the problem

A dynamic library is loaded when the program is running. It means the binary code of a third-party function is not linked when building the program, but only when this function is called. As the OS can not search for a library anywhere each time a third party function is called at run time, the dynamic libraries have to be located in some predefined directories. For example for Unix-like systems : "/lib", "/usr/lib", "/usr/share/lib" [2].

For example, the CMocka library may have been installed in "/usr/local/lib" (you can check) and sometimes this repertory is not one of the predefined library directories. In order to add it to this list, follow the next instructions.

B.2 Here is the solution of the problem

As you will change some protected file, you need the admin (root) rights. Adding "sudo" at the beginning of a command enables to run this command with admin rights.

Type this to edit the configuration file with the predefined directories to search for dynamic libraries :

```
$ sudo gedit /etc/ld.so.conf
```

Suppress nothing in this file and add a new line with :
/usr/local/lib
Save the file, quit *gedit*. Run the library manager to take into account your changes:

```
$ sudo ldconfig
```

Quit the app from where you launch your program (Codeblocks, terminal,...) and re-run it so it can benefit from the new configuration.

[1] Try to install in predefined directories for dynamic librairies to avoid this problem and to keep a clean system.
[2] We insist : install the library where it is recommended for your OS.

ANNEXE C

gcov/lcov

gcov aims to measure your code coverage. *lcov* is a graphical front-end for GCC's coverage testing tool *gcov*.

C.1 Before running gcov

To use this *gcc* tool, the compiler and the linker need to manage additional information in the binary code of the program. To generate a flow graph of a program and exploit it to generate profiling information when running the program, you need to add "-fprofile-arcs -ftest-coverage" or "-coverage" in the compiler and linker options. In a Makefile, it would be in both CFLAGS and LDFLAGS variables. For a CodeBlocks project:

- for the compiler : go in the Build options ⇒ Compiler Setting ⇒ add *-fprofile-arcs -ftest-coverage* in the "Other options" window.
- for the linker : go in the Build options ⇒ Linker Setting ⇒ add *-fprofile-arcs -ftest-coverage* in the "Other linker options" window.

After this configuration change, do not forget to clean your project, re-build it and run it again.

C.2 How to interpret gcov results

C.2.A Files generated by gcov

When running the test with such a configuration, two types of files are generated :

- *.gcno* files generated when the file is compiled to build the flow graph
- *.gcda* files generated when running the program. It will deliver counts for each code line. This counter is incremented by one each time the code line has been executed when running the program. As *gcov* can be used to profile a code, not only a test code, the counts are cumulative. It means that if you run the program another time, the count will be added to the results of the previous runs. It can be be very useful when you want to profile your code to know where it should be improved or optimized. However, if you do not want to cumulate the count (which is our case since we only want the coverage of the tests), you have to remove the *.gcda* files before running the test.

Those files are generated in the same directory than the objects files, for example "obj/Debug" in a classical CodeBlocks project. Once you are in the directory with gcda and gcno files, you type for example to see the coverage of the "counter.c" code after running your tests :

```
$ gcov counter.c
```

It will generate a *.c.gov* file. In this file, you will found the count for every code line:

- a number representing how many times this line has been executed if so
- - means this line has not been executed but it is not a relevant code line (blank line, comment,...)
- ##### means this line has not been executed whereas it seems to be a relevant code line

You will also find :

- for a function, how many times it has been called, how many times it has returned something and what percentage of its blocks were executed
- for a branch (if, for, while,...), if it has been taken and the percentage of choice for each branch.
- for a call, a flag to show if it was executed at least once and the number of times the call returned something.

C.2.B How to interpret gcov results using lcov

If *lcov* is not instaler, type :

```
$ sudo apt-get install lcov
```

Then type

```
$ lcov --capture --directory projectdirectory --output-file coverage.info
$ genhtml coverage.info --output-directory out
```

where *projectdirectory* is the directory with the files generated by *gcov* and *out* is the directory where the file *index.html* will generated. Open this latter to see the result of your tests.