Piscine — Tutorial D5

version #b4173ffc5996c36bf7f9730185ebc8357500bd86

Assistants C/Unix 2024 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

*https://intra.forge.epita.fr

1 Assert

The `assert(3)` *macro* allows you to perform runtime checks in your program. If the given expression evaluates to false, the program will crash and print various information. This macro will always be allowed and can be really useful, so it is in your best interest to know how to use it properly.

`assert(3)` is defined in the `assert.h` header, and you can use it as shown below.

```c
#include <assert.h>
#include <stdio.h>

int div(int a, int b)
{
    assert(b != 0);
    return a / b;
}

int main(void)
{
    printf("div(10, 2) = %d\n", div(10, 2));
    printf("div(5, 0) = %d\n", div(5, 0));
    return 0;
}
```

Let us run the above program. It will abort if the condition is not met.

```
42sh$ gcc -g -Wall -Wextra -Werror -Wvla -pedantic -std=c99 example1.c -o example1
42sh$ ./example1
div(10, 2) = 5
example1: example1.c:6: div: Assertion `b != 0' failed.
[1]    10722 abort (core dumped)  ./example1
```

The first call to `div` runs fine, however the second one crashes.

The output message is really self-explanatory:

- `abort (core dumped) ./example1` means that the program `./example1` aborted.
- `example1: example1.c:6: div:` provides you with the exact location of the crash: in the `example1` process, at line 6 in the `example1.c` file and in the `div` function.
- Finally `Assertion 'b != 0' failed.` directly displays the failed assertion.

The condition in the assert statement is printed when the assert fails. You can play with it to get some additional information displayed in the error message. For example, add `'&& mystring'` to your condition:

```c
#include <assert.h>
#include <stdio.h>

int fact(int n)
{
    int res = 1;

    assert(n >= 0 && "The factorial function is only defined for positive numbers");
```

(continues on next page)

(continued from previous page)

```
    while (n)
        res *= n--;

    return res;
}

int main(void)
{
    printf("fact(5) = %d\n", fact(5));
    printf("fact(-2) = %d\n", fact(-2));
    return 0;
}
```

```
42sh$ gcc -g -Wall -Wextra -Werror -Wvla -pedantic -std=c99 example2.c -o example2
42sh$ ./example2
fact(5) = 120
example2: example2.c:9: fact: Assertion `n >= 0 && "The factorial function is only defined␣
↪for positive numbers"' failed.
[1]    8140 abort (core dumped)  ./example2
```

Going further...

As explained in the `assert(3)` manual, the macro does nothing when `NDEBUG` is defined (with `gcc -DNDEBUG`). This feature enables avoiding the check *overhead* once your code is deployed.

It should be set in the `CPPFLAGS` variable of your Makefile.

Be careful!

You should not have side effects in your assert statements, otherwise the behavior of your program might change depending on whether assertions are enabled!

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

int main(void)
{
    int i = 1;
    assert(i++ > 0);
    assert(i++ > 1);
    assert(i++ > 2);
    printf("%d\n", i);

    return (0);
}
```

```
42sh$ gcc -g -Wall -Wextra -Werror -Wvla -pedantic -std=c99 example3.c -o example3
42sh$ ./example3
4

42sh$ # Now run again with -DNDEBUG to disable assertion checks
42sh$ gcc -g -Wall -Wextra -Werror -Wvla -pedantic -std=c99 example3.c -o example3 -DNDEBUG
42sh$ ./example3
1
```

2 Unit testing

2.1 Introduction

When programming, you will inevitably create *bugs*. This is why we test our programs with various inputs and tricky cases, but doing so manually is rather tedious and error-prone. Imagine switching constantly between exercises or projects and having to remember which tests you used last time, how to input them to your program, what the expected values were, ...

Today we will learn how to automate the testing process and focus on one type of tests: **unit tests**.

> *"Unit tests are typically automated tests written and run by software developers to ensure that a section of an application (known as the "unit") meets its design and behaves as intended."*
>
> ---Wikipedia

As the name suggests, unit tests are focused on a specific portion of your code (for example one function), not the whole program. The idea is to make sure small and individual parts of the code work as expected and are tested thoroughly.

During your first ING1 projects, we will talk plenty about tests intended to verify the entire program execution in addition to only one function: these are called **functional tests**. For the *piscine*, we want you to focus on unit tests since they are particularly adapted to the programming exercises we give you (most of the time you have a small number of functions to implement).

Be careful!

You **must** write unit tests for each exercise you attempt to solve. Teaching assistants will make sure you have some tests before helping you debug your code.

Going further...

You can integrate your unit tests with your programming workflow:

- Version the test files using Git.
- Add a `make check` rule in your Makefile to build and run your test suite.

2.2 Example

Here is a simple function used to compute a^b.

```c
int my_pow(int a, int b)
{
    int res = 0;

    for (int i = 0; i < b; i++)
    {
        res += a;
    }

    return res;
}
```

At first glance, this function might seem alright. Let us try with different assertions to check our function return values:

```c
#include <assert.h>

int main(void)
{
    assert(my_pow(1, 2) == 1);
    assert(my_pow(2, 4) == 16);

    return 0;
}
```

```
42sh$ gcc -Wall -Wextra -Werror -Wvla -pedantic -std=c99 my_pow.c -o my_pow
42sh$ ./my_pow
my_pow: my_pow.c:18: main: Assertion `my_pow(1, 2) == 1' failed.
[1]    8989 abort      ./my_pow
```

Unfortunately it fails, which means our `my_pow` function contains one or more bugs.

Looking back at the code, there are two bugs: one is inside our loop because we need to multiply values not add them, the other one is the initialization of `res` which needs to be set to 1 and not 0.

```c
int my_pow(int a, int b)
{
    int res = 1;

    for (int i = 0; i < b; i++)
    {
        res *= a;
    }

    return res;
}
```

Let us recompile the program.

```
42sh$ gcc -Wall -Wextra -Werror -Wvla -pedantic -std=c99 my_pow.c -o my_pow
42sh$ ./my_pow
42sh$
```

Everything passed! Now we can add more complicated cases, like what will happen if the user gives a negative number, or `INT_MAX` as a parameter?

2.3 Testing frameworks

You *could* use `assert(3)` for your exercises test suites, however you will quickly understand that it is not the most efficient way of doing. It suffers from some flaws:

- Unless you generate one binary per test, failing a test that is done using `assert` will result in the whole test suite stopping, since `assert` works by stopping the execution of the program immediately.
- `assert` only works if you already have a boolean expression and will only output the line where the assert failed. No runtime-dependent information will be displayed such as the value that was returned by a called function.
- If test reports are needed, they must be done either by hand or by using a library and calling it explicitly after each test.

Those are some reasons that lead to the creation of dedicated **testing frameworks**.

This year, you will learn to use the Criterion unit testing framework.

3 Criterion

Although many testing frameworks exist, we will only see how to use Criterion this semester. Reasons for this include:

- Test isolation means that if a test makes the program crash, the testing framework will continue working and simply report that the specific test crashed. In the same way, global variables modified in a test will only affect that test and the value will not have changed when running the next tests. Signal handling can also be tested thanks to that.
- Custom abort messages can be written using `printf`-like formatting. Standard error messages already take advantage of that and will for instance show both the expected and obtained values when comparing two integers.
- The possibility to export test results in a custom format. XML files and TAP are already implemented out of the box.
- Cases where a program is expected to exit can be tested.

3.1 Getting started

Let us rewrite our previous tests for `my_pow` using the Criterion library.

The first thing to do when writing a new test suite using Criterion is to include its header: `criterion/criterion.h`.

Once included, you can try compiling your test suite without writing a `main` function (Criterion provides one by default). A *linker* flag must be used to let the compiler know you need this external library to build your code: `-lcriterion`.

```c
#include <criterion/criterion.h>

int my_pow(int a, int b)
{
    int res = 1;

    for (int i = 0; i < b; i++)
    {
        res *= a;
    }

    return res;
}
```

```
42sh$ gcc -Wall -Wextra -Werror -Wvla -pedantic -std=c99 my_pow.c -o my_pow -lcriterion
42sh$ ./my_pow

[====] Synthesis: Tested: 0 | Passing: 0 | Failing: 0 | Crashing: 0
```

From there you can add tests in your file like so:

```c
Test(suite_name, test_name)
{
    // Your test goes here
}
```

Going further...

As you can see, tests are regrouped under *test suites*. Those are defined automatically by Criterion when you create a test, but you can also choose to create them manually if you want to configure precisely how your test will run. However, we will not cover user-defined test suites in this tutorial.

3.2 Criterion assertions

Many assertion functions are provided by Criterion. The most useful one is `cr_assert_eq` and `cr_expect_eq`:

```c
Test(Basics, simple_test)
{
    int actual_value = 0;
    int expected = 1;
```

(continues on next page)

(continued from previous page)

```
    cr_expect_eq(actual_value, expected);
}
```

It compares its two parameters and marks the test as failed if they are not equal. Try running this test. You will notice that Criterion automatically formatted the error message to show which assertion failed.

Going further...

Many other assertions are available, which are listed on Criterion documentation. One example is `cr_expect_eq`, which works similarly to `cr_assert_eq` but does not abort the current test on failure while still marking it as failed.

It is also possible to add a custom error message in case the assertion fails:

```
Test(Basics, formatting_test)
{
    int actual_value = 0;
    int expected = 1;
    cr_expect_eq(actual_value, expected, "Expected: %d. Got: %d", expected,
                 actual_value);
}
```

If you want your test to also have some logging messages, you can use the `cr_log_info`, `cr_log_warn` and `cr_log_error` functions, like so:

```
Test(Basics, logging_test)
{
    cr_log_info("This is an informational message. They are not displayed "
                "by default.");
    cr_log_warn("This is a warning. They indicate some possible malfunction "
                "or misconfiguration in the test.");
    cr_log_error("This is an error. They indicate serious problems and "
                "are usually shown before the test is aborted.");
}
```

You might also want to check the standard and error output to ensure that your program is correctly working. You can use the `cr_assert_stdout_eq_str` and `cr_assert_stderr_eq_str` assertions, like so:

```
#include <criterion/criterion.h>
#include <criterion/redirect.h>
#include <stdio.h>

void error()
{
    fprintf(stderr, "This is an error message\n");
}

Test(Basics, error_case)
{
    cr_redirect_stderr();
    error();
```

(continues on next page)

(continued from previous page)

```
    fflush(NULL);
    cr_assert_stderr_eq_str("This is an error message\n");
}
```

Getting back to our example, here are our previous `my_pow` unit tests converted to Criterion tests:

```
Test(my_pow_testsuite, one_to_the_power_of_two)
{
    int actual = my_pow(1, 2);
    int expected = 1;
    cr_expect_eq(actual, expected, "Expected %d. Got %d.", expected, actual);
}

Test(my_pow_testsuite, two_to_the_power_of_four)
{
    int actual = my_pow(2, 4);
    int expected = 16;
    cr_expect_eq(actual, expected, "Expected %d. Got %d.", expected, actual);
}
```

```
42sh$ gcc -Wall -Wextra -Werror -Wvla -pedantic -std=c99 my_pow_test.c my_pow.c -o my_pow_
↪test -lcriterion
42sh$ ./my_pow_test

[====] Synthesis: Tested: 2 | Passing: 2 | Failing: 0 | Crashing: 0
```

You can see the output is fancier than using `assert(3)`. Here is the output when a test fails:

```
42sh$ ./my_pow

[----] my_pow.c:19: Assertion failed: Expected 1. Got 0.
[FAIL] my_pow_testsuite::one_to_the_power_of_two: (0.00s)
[====] Synthesis: Tested: 2 | Passing: 1 | Failing: 1 | Crashing: 0
```

Going further...

Take a look at the usage documentation (run your binary with the `--help` option). Some options can come in handy, such as `--verbose` to control the amount of information printed or `--jobs N` to run tests using `N` concurrent jobs.

3.3 Criterion debug

Have you ever tried to debug a binary compiled with criterion ? In order to debug your program with a criterion testsuite you can use the argument `--debug=gdb`. Which will start a gdb server.

```
42sh$ ./my_pow_test --debug=gdb
[----] Criterion v2.4.1
[====] Running 2 tests from my_pow_testsuite:
Process /home/acu/my_pow_test created; pid = 53849
Listening on port 1234
```

Then in another terminal run a gdb instance.

```
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
Reading /home/acu/my_pow_test from remote target...
```

You are now able to place `breakpoints` and print information. To debug the next test, use `continue` until the process you are debugging ends.

```
(gdb) continue
[Inferior 1 (process 53849) exited normally]
```

You can noticed that the first test passed and criterion launched a new gdb server for the second test

```
[----] Criterion v2.4.1
[====] Running 2 tests from my_pow_testsuite:
Process /home/adrien/my_pow_test created; pid = 53849
Listening on port 1234
Remote debugging from host ::1, port 35446
[RUN ] my_pow_testsuite::one_to_the_power_of_two
[PASS] my_pow_testsuite::one_to_the_power_of_two: (0,00s)

Child exited with status 0
Process /home/acu/my_pow_test created; pid = 54125
Listening on port 1234
```

You can run again the command `target remote localhost:1234`. Congratulations, you are now debugging the second test.

3.4 Additional features

You should notice in the previous output that one test failed, however the program did not crash and our second test executed correctly. This is called **test isolation** and is a feature of Criterion as well as other testing frameworks.

Criterion has many advanced features that might be interesting to use in your own test suite: parametrized tests, testing report hooks, theories, catching signals, testing exit code, tests code coverage, ...

Tips

Again, you **should** take some time to read the Criterion documentation. It will be useful throughout the semester.

4 Exercises

4.1 My atoi_base

4.1.1 Goal

You have to implement the following function:

```
int my_atoi_base(const char *str, const char *base);
```

This function must have the same behavior as the `atoi(3)` function, but in a specified base.

`str` is a string and represents a number in the base `base`. `str` must be converted into the associated decimal value.

4.1.2 Examples

```
my_atoi_base("ff", "0123456789abcdef");
```

must return the value 255.

```
my_atoi_base("-ff", "0123456789abcdef");
```

must return the value -255.

```
my_atoi_base("77", "01234567");
```

must return the value 63.

```
my_atoi_base("WQWW", "QW");
```

must return the value 11.

If one of the digits of the number to convert is not included in the given base, the result must be zero.

The `str` string must follow the following format in specified order:

- A possibly empty sequence of whitespace characters that will be discarded
- An optional single plus or minus sign
- A sequence of digits included in `base` string

If `str` does not match this format, you must return 0.

The string `base` always represents a valid base. If the `str` string is empty, you must return 0 (zero).

4.2 My itoa_base

4.2.1 Goal

You have to implement the following function:

```c
char *my_itoa_base(int n, char *s, const char *base);
```

This function should convert the integer `n` in its representation in base `base` and store this representation in `s` (without forgetting to end it by '`\0`'). The function returns the resulting string (the same as the one given by the argument `s`). Consider that the caller already allocated the space needed in `s`.

You need to handle negative numbers only in base 10.

The argument `base` is interpreted as follows: the `i`-th character of the string is the representation of the value `i` in the target base.

Base 1 is the only base you can leave behind. All other bases must be handled.

4.2.2 Examples

```c
my_itoa_base(42, s, "0123456789ABCDEF");
```

s will be equal to "`2A`".

```c
my_itoa_base(32, s, "0123456789abcdef");
```

s will be equal "`20`".

```c
my_itoa_base(12, s, "01");
```

s will be equal "`1100`".

```c
my_itoa_base(80, s, "0123456");
```

s will be equal "`143`".

I must not fear. Fear is the mind-killer.