Experiment 3 – Testing Library Robustness

This task can only be solved with Unix or Linux. You have to implement a robustness test for the glibc function **`fputs`**. The robustness is tested for common and uncommon argument values of that function.

Contents

Robustness

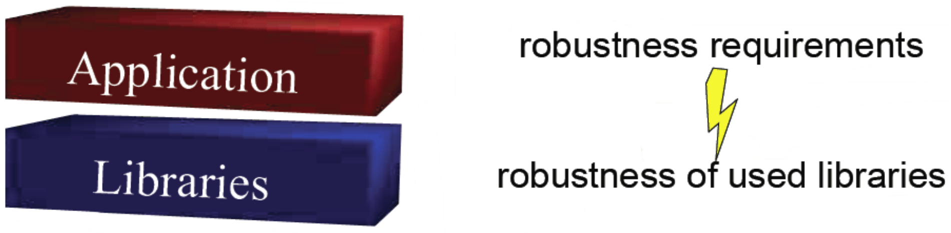
Before we talk about finding robustness issues we have to define the term robustness. In short: A function is robust if it behaves complaisant for erroneous input.

The robustness problem

Each real application needs to behave robust. This means there are general requirements for unspecified inputs:

- „do not crash“
- „do not lose consistency“
- ...

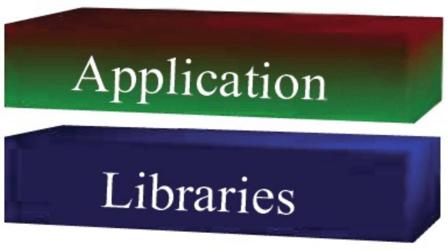
In other words, the program shall not crash or come into an inconsistent state or face any other problems only because it has been fed with unexpected input.



Developers might be forced to use some libraries, with unknown robustness. The source code is not accessable in the general case, so source code inspection is not an option. Developers need another approach to "measure" the robustness of libraries. This is the topic of the current task.

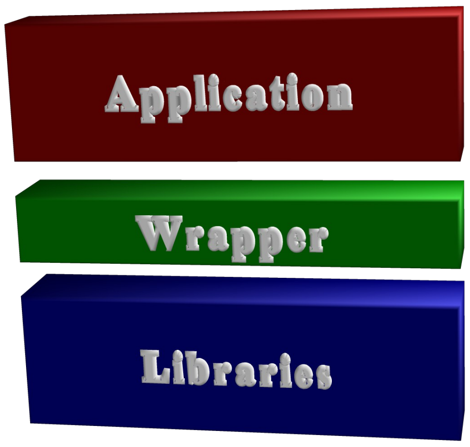
Another task will deal with non-robust libraries by introducing a patching wrapper. Here comes a short introduction:

- First option is to increase robustness in your application.



```
if (p1 != NOT_ALLOWED)
    foo (p1);
else
    throw SomeError (...);
```

- Second option is to add a robustness wrapper. This is a special library that treats any unspecified input and passes the others to the original library.



```
void foo(int p1) {
    if (p1 != FORBIDDEN)
        original_foo (p1);
    else
        throw SomeError (...);
}
```

Finding robustness offenses

Robustness is commonly determined per module. For libraries modules are exported functions. To measure a library's robustness, the robustness of each exported function must be determined.

In theory a function must be tested against each possible combination of input values. In practice it is sufficient to select common and uncommon representatives for each data type. We will deal with pointer types. Pointers are often used in an non-robust way. So it makes sense to test all functions with pointers as arguments. Good representatives for pointers are:

- `NULL`
- pointer to a read only area
- pointer to a write only area
- pointer to a read/write area

Pointer have often a certain semantic. That's why the content of the memory area a pointer points to might be important. For strings it could be:

- `\0` terminated string
- string without `\0` termination

For files:

- handle to read only file
- handle to write only file
- handle to already closed file
- pointer to an invalid file handle

Both sets of semantical depending pointer representatives must be combined with the common pointer representatives.

Beside the task to generate values to test on, you have to make sure that a crash of the function to test does not crash your test itself. To run the function to test in a child process is one good option. You fork a process. In the child you only run the test function with generated test values. The parent observes the child process (termination, crash, etc).

Your Task

You have to write a tool for testing the C-Library function **`fputs`**.

- **1. step:** building a test routine for a single test on `fputs`
- **2. step:** write functions for generating test values for each needed data type

Once you have that done the next task could be to build a robustness wrapper for brittle library functions.

1st step: Building a test routine

A test routine executes a specific library function with a particular input, e.g.:

```
abs (12);
```

It must detect if execution does not terminate, crashes or if the function returns unspecified return values.

Your task is to build this routine for the function:

```
int fputs (const char* s, FILE* stream);
```

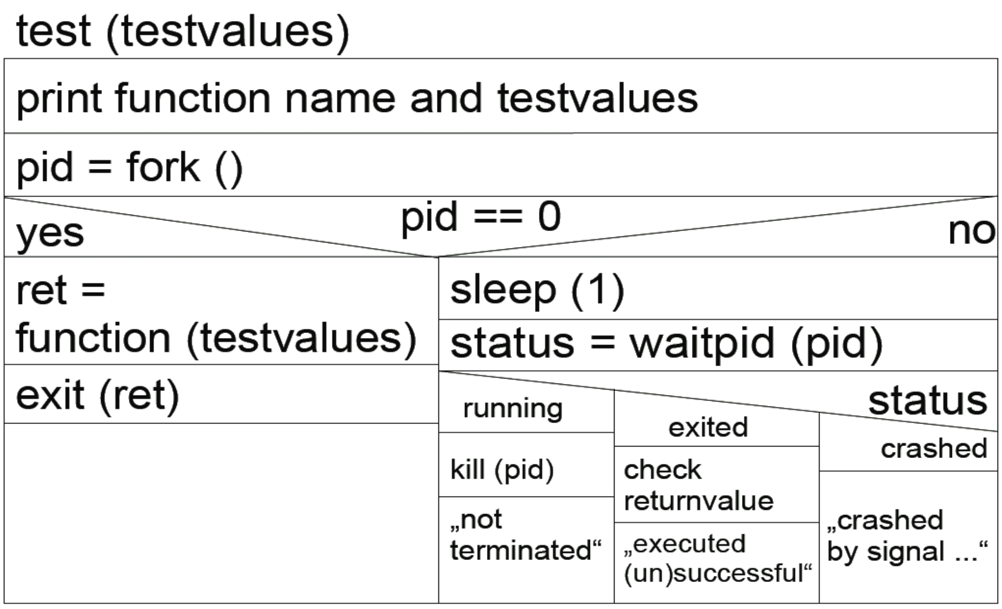
Your test routine receives a **`TestCase`** structure, that describes a value for each parameter of **`fputs`**:

```
void test_fputs (TestCase& cstr, TestCase& file) {...}
```

Here are some hints that might help you solve this task:

- Execute the function to test (**`fputs`**) in a separate process: use **`fork ()`** to create a new process. The process should exit with the return value of the function.
- Then wait for a little (approximately 0.1 second).
- Use **`waitpid (..., WNOHANG...)`** to request the status of the child process.
- Print an evaluation of the requested status to the screen.
- Read the man pages to **`fputs`**, **`fork`**, **`waitpid`** and **`sleep`**...

The following Nassi-Shneiderman diagram depicts the algorithm of your test function:



2nd step: Generating test values

Function **`test_fputs (...)`** from 1st step tests exactly one C-library function for one combination of possible input values.

So now generate a list of test values for each parameter type to be used. Examples:

- **`FILE*`**: should be a handle for a read-only-opened file, ... for a write-only-opened file, ... for an already closed file, **`NULL`**, pointers to invalid file handles, ...
- **`const char*`** `(string)`: valid (`NULL` terminated) string, invalid string – not `NULL` terminated, **`NULL`**
- most values (like file handles) must be generated via a short generator function

Structure **`TestCase`** in `tests.h` describes a test representative:

- an ID
- a describing string
- the expected return value (not used)

We have specified 12 test cases for **`FILE*`** and 8 for **`const char*`**.

You will have to write two function `generateCSTR(int)` and `generateFILE(int).` Each of both takes an ID and generates the corresponding test value. File handles should work on a copy of the test original file, because a test might destroy the file represented by the file handle.

Summary

Write a program which tests the function **`int fputs (const char*, FILE*)`** for robustness against specified and unspecified parameter values. The representative are specified by us in `tests.h`. Write generator functions to generate concrete representative values. Tie it all together in `main(...)`.

Use our API for recording your test results (see `stats.h`). You should also use the function from `tools.h` for test value generation.

Hints

We provide you with the specification of the test representatives, some helper functions for test value generation and some functions for statistical analysis.

Read the documentation in the comments of the given files.

Test representatives

The test representatives are specified in `tests.h`. You have to generate concrete test value for each of them. The two types are of course: **`const char*`** and **`FILE*`**. You must test all possible combinations of test values to pass our checks. If your are unsure about the meaning a concrete test representative specification, use the forum or ask us.

Summary of file handle test representatives:

- `NULL` pointer
- file opened read-only, write-only, read/write
- file already closed
- pointer to valid file structure (memory read-only, write-only, read/write)
- pointer to a `NULL` page (read-only, write-only, read/write)
- pointer to inaccessible memory

Summary of string test representatives:

- `NULL` pointer
- pointer to `NULL` terminated string (read-only, write-only, read/write)
- pointer to not `NULL` terminated string (read-only, write-only, read/write)
- pointer to inaccessible memory

With all combinations you have 96 test cases.

Helper Functions

We provide you with some helper functions. The are declared and documented in `tools.h`. Function **`malloc_prot`** could be used to reserve memory as read-only or write-only. A crashed test could destroy a file. That's why you must copy your test file before opening it for generating a **`FILE`** test value. You can use **`filecopy`** to this. Generating some test values may need a memory page of zeros. This is provided by **`NULLpage`**. Finally the function **`sleep`** is a wrapper for **`nanosleep`**. You can delay the execution of your test (while waiting for the child process) for fractions of a second.

The use of these functions will not be enforced. You do not have to use them.

Statistical Analysis

The functions declared and documented in stats.h fulfill two objectives:

- provide you with an easy API to record and analysis
- gives us a way into your test program to check it

That's why there use is obligatory. Make sure that you use them as documented. There are three kinds of functions:

- **`record_start_test_fputs (...)`** must be called at the start of each test; it prints out a test description depending on the given test representatives
- **`record_*_test_fputs (...)`** must be called to record each test result; they print out a short message and store the results for a summary; depending on your test result, call the appropriate function (**`ok`**, **`crashed`**, **`stopped`**, or **`timedout`**)
- **`print_summary()`** must be called at the end of your program; it prints out a short summary of your tests

Side Effects

Test value generation is not side effect free. That's why the order of calling `generateCSTR` and `generateFILE` matters. Our test system uses this order:

1. `generateCSTR`
2. `generateFILE`

Please stick to this order to pass our tests. In the following we present the details of this side effect. The tests who's results differ depending on the order are:

- pointer to valid string (memory read only), FILE already closed
- pointer to valid string (memory read/write), FILE already closed

Function `fclose(FILE*f)` calls `free(f)` on the file handle passed to it. A succeeding call to a LibC function might reallocate that memory for other purpose. In this case the content of the memory reference by `f` is invalid (overwritten) and `fputs` will crash. But as long as the memory is untouched `fputs` can read it and will return with an error code.

If the generator functions are executed in the first order (as in our reference implementation) `fclose` is the last call to a LibC function before `fork` and `fputs` (in the child). No memory is allocated in between. Function `fputs` can access the structure behind `f` and does not crash.

The execution in the other order (the bad one ;-) executes between the `fork` and `fputs`, `malloc_prot` (in `generateCSTR`). Function `malloc_prot` itself makes use of several LibC functions. Among others it allocates memory. So the memory referenced by `f` is invalid. And `fputs` crashes.

Further reading

For further information look to the Ballista project, our SFT lecture or the next task.

- Ballista
  - if you want more detailed information: http://www.ece.cmu.edu/~koopman/ballista/
  - testing of the POSIX API under different operating systems
  - evaluates and compares robustness of these operating systems and their POSIX implementation
- Lecture
  - “Software Fault-Tolerance”

Organizational Remarks

Deadline

Solve this week's experiment at home or in the lab.

Please be aware that there is a deadline for this task. You can find the exact date published on our website. If you do not hand in your solution until the specified day we will regard your solution as missing so you will not receive your certificate.

Directory Structure

We require a specific directory structure for your solutions.

```
robustness/
    Makefile              - Makefile
    tools.h               - specifies helper functions for test value
generation and test execution
    tools.cpp             - implement functions of tools.h
    stats.h               - specifies statistics function for test results
analysis and recording
    stats.cpp             - implement functions of stats.h
    tests.h               - specifies the test representatives
    test.txt              - a file to test on (run tests only
on copies of it!!!)
    fputs_test.cpp        - your test of fputs
```